Universitat Politècnica de Catalunya

Escola d'Enginyeria de Manresa

Treball Final
del
Grau en Enginyeria de Sistemes TIC

Disseny d'un servei de computació paral·lela i distribuïda

Esteve Martin

6 de juliol de 2018

Director: Sebastià Vila-Marta

Resum

L'objecte d'aquest treball és explorar i experimentar amb la programació paral·lela en *clústers* de computadors. Aquest model de programació és, segurament, un dels que ha resultat més fructífer.

El treball inclou una recopilació de l'evolució tecnològica en l'àmbit de la computació paral·lela, fent un èmfasi especific en l'estàndard MPI.

Per tal d'assolir l'objectiu del treball, s'ha comptat amb un *clúster* de plaques Raspberry Pi, amb processadors d'arquitectura ARM, el qual es mostra com configurar i utilitzar. L'objectiu principal del treball és aconseguir realitzar proves sobre el *clúster* i treure'n resultats per avaluar-ne el rendiment i també verificar el funcionament de les tecnologies utilitzades.

Abstract

In this project we explore and test some of the most influential technologies related to parallel computing. In a first part, a short compilation of this technologies is made, explaining their history and some basics about how they work. We can highlight among them the MPI standard, which we use extensively in this project.

In order to use these tchnologies, we have set up a cluster made of Raspbery Pi boards, which use processors of ARM architecture. In this same document we show how to configure and use this cluster. The main objective of the project is to be able to carry on some tests on the cluster, so that we can verify and evaluate the performance of both the cluster and the stack of software used with it.

Índex

Índex de figures

Índex de taules

1 Introducció

Moltes branques de la ciència i la tecnologia requereixen de la computació per a resoldre alguns dels seus problemes rellevants. La previsió del clima, l'anàlisi de reaccions de partícules o la recerca en les matemàtiques en són alguns exemples. Malauradament, la potència de càlcul dels computadors convencionals és massa escassa per a aquestes tasques: el temps que triguen a calcular en molts casos és inviable.

Independentment d'això, el sector electrònic s'ha trobat que per diversos factors, principalment la generació de calor, cada cop resulta més difícil augmentar la freqüència dels processadors.

Aquestes dues problemàtiques han mogut l'interès de la comunitat informàtica cap a la programació paral·lela, la qual permet de forma eficaç i escalable multiplicar el rendiment d'un sistema. Molts països i institucions han optat per desenvolupar supercomputadors massius i oferir-los com a servei. A part d'això moltes organitzacions també han adaptat arquitectures paral·leles de menor escala per altres propòsits.

Aquest projecte té com a objectiu explorar algunes de les tecnologies utilitzades actualment en el món de a computació paral·lela per així comprendre millor com funcionen aquests «monstres» que són els supercomputadors. Per tal de dur a terme aquesta tasca s'ha dividit el projecte en tres parts, cadascuna amb les seves fites. En la primera part s'ha cercat informació i s'ha recopilat una col·lecció de les diferents tecnologies que s'han fet servir en aquest àmbit. Seguidament, s'ha muntat i configurat un clúster de computadors Raspberry Pi, per tal de poder posar en acció algunes de les tecnologies vistes en l'apartat anterior. Per finalitzar, s'han fet algunes proves en el clúster per verificar el seu funcionament.

2 Estat de l'art

En aquest capítol s'inquireix sobre la computació paral·lela i algunes de les eines que es basen en ella. Per una banda és mostra breument la història d'aquest model de programació i com s'ha utilitzat en *clústers* de computadors. També es parla sobre els supercomputadors, màquines immenses que aprofiten aquest model i que tenen molta presencia en l'actualitat.

Per altra banda, es presenten algunes de les tecnologies més utilitzades en aquest àmbit, per avaluar quines d'elles són les més adients per instal·lar en el nostre *clúster*. Podem destacar l'estàndard MPI i l'estructura de *clústers* Beowulf.

Per acabar, es presenten alguns computadors actuals que utilitzen les tecnologies descrites prèviament, per així obtenir una visió més actualitzada de la computació paral·lela.

2.1 Història dels clústers i la computació en paral·lel

En el camp de la informàtica, un clúster es refereix a un grup d'ordinadors connectats entre si en una xarxa i que mitjançant un software determinat actuen com si fossin un sol sistema.

A [14], PFister ens explica que el terme *clúster* va aparèixer al voltant de la dècada dels 60; no tant promogut per cap organització ni companyia sinó per els usuaris corrents, que no en tenien prou amb un sol ordinador i n'adquirien un altre per realitzar la seva feina. Aquests primers clústers, però, eren un concepte molt difús i sovint es tractaven de màquines addicionals dedicades a emmagatzemar backups o altres arxius que no cabien a la màquina principal.

El concepte de *clúster* com a un conjunt de màquines que executen feina paral·lela s'atribueix a Gene M. Amdhal, treballador d'International Business Machines (IBM), qui l'any 1967 va publicar el que es considera el primer article sobre computació paral·lela, [1]. En aquest article parlava, des d'un àmbit matemàtic, de la possibilitat d'accelerar l'execució de les tasques mitjançant la divisió i paral·lelització d'aquestes, ja sigui per a *clústers* o per sistemes multiprocessador. Un dels conceptes més rellevants d'aquest article és la llei d'Amdahl. Aquesta llei permet predir el màxim increment en velocitat (*speedup*) que podem obtenir segons el nombre de nodes paral·lels N i el percentatge de codi que tenim paral·lelitzat, P, mesurat en tant per u. Vegi's la figura 2.1. La fórmula que descriu aquest comportament és la següent:

$$Speedup(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

Aquest treball va tenir una gran influència en el creixement de la tecnologia de *clústers*. A partir d'aquest moment els *clústers* van veure una evolució gradual fortament lligada a l'aparició de noves tecnologies de xarxes i al desenvolupament del sistema operatiu UNIX.

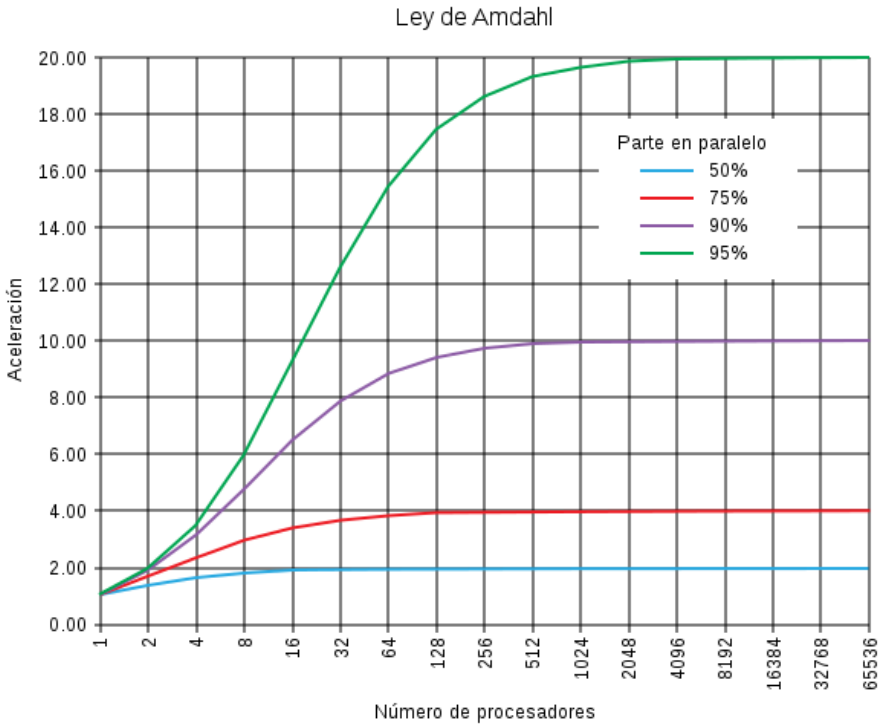


Figura 2.1: Corbes descrites per la llei d'Amdahl

Font: `https://es.wikipedia.org`

El primer sistema dissenyat com un *clúster* en tot el sentit de la paraula va ser el Burroughs B5700 que va aparèixer cap a la meitat de la dècada dels 60. Era un sistema que disposava de 4 ordinadors lligats per un disc d'emmagatzematge comú el qual feien servir per distribuir-se la càrrega de treball.

L'any 1969 el projecte The Advanced Research Projects Agency Network (ARPANET) va aconseguir comunicar un seguit d'ordinadors utilitzant una xarxa econòmica i simple, juntament amb la primera implementació del protocol Transmission Control Protocol/Internet protocol (TCP/IP). Tot i que el sistema resultant d'aquest projecte no és en sí un *clúster*, ja que els ordinadors no executaven codi paral·lel, ARPANET va suposar de totes maneres una revolució en l'àmbit de la computació paral·lela gràcies a que va estandarditzar l'us de paquets per les comunicacions entre processos. Cal dir també, i com a curiositat, que el projecte ARPANET va seguint creixent fins a convertir-se en els ciments de L'Internet tal i com avui el coneixem.

El primer producte comercial per a *clústers* va sorgir el 1977, es tracta de l'ARCnet, desenvolupat per Datapoint. Arcnet era un protocol que mitjançant una comunicació per xarxa local permetia a micro-computadors obtenir accés compartit a un sistema de disquets. El producte no va tenir molt èxit ja que en aquells temps els *clústers* encara no havien esdevenint una tecnologia popular i desitjada.

L'any 1984 va aparèixer el VAXCluster de Digital Equipment Corporation (DEC). És tractava d'un conjunt d'ordinadors corrent el sistema operatiu VMS ara conegut com a OpenVMS. Aquest va ser el primer clúster en aparèixer que permetia executar codi paral·lel. A part d'això també

disposava de *filesystem*, sistema de seguretat i sistema de permisos compartits a nivell de *clúster*. VAXCluster té prestigi de ser l'invent que va popularitzar els *clústers*. No va ser fins a partir de las seva aparició que els *clústers* i el processament compartit es van obrir realment al gran públic.

Els següents anys ens van portar nombrosos avenços en la tecnologia de *clústers*, destaca la dècada dels 90 al voltant de la qual van aparèixer algunes de les tecnologies més influents actualment com són la Parallel Virtual Machine (PVM) el 1989, el *clúster* Beowulf el 1994 i l'estàndard MPI al mateix any.

2.1.1 PVM

PVM és una conjunt d'eines software i llibreries que permeten a un grup d'ordinadors en una xarxa connectar-se per formar un únic computador virtual que combina la capacitat de processament dels seus membres.

Al llibre *PVM: Parallel Virtual Machine*, [11], se'ns expliquen totes les característiques i peculiaritats d'aquesta eina. Segons es descriu al llibre, la computació de PVM es basa en tasques, es generen processos paral·lels que es poden executar en una o més màquines i que es poden comunicar per tal de cooperar entre ells. Mitjançant PVM l'usuari pot triar executar una tasca en un conjunt de computadors que escull lliurement. A més, també se suporten les arquitectures de processador múltiple de manera que es poden tenir diferents tasques executant-se en paral·lel en un únic computador.

PVM funciona mitjançant un dimoni anomenat `pvmd3` que s'encarrega tant de gestionar la màquina virtual que engloba els diferents nodes participants com d'habilitar la comunicació entre processos. Per tal a poder fer aquesta feina, cadascun dels ordinadors col·laboradors executa una copia del dimoni. PVM també inclou una llibreria que ofereix el conjunt de rutines per tal que un procés pugui comunicar-se amb el dimoni i per a gestionar la xarxa de nodes que conformen el computador virtual.

2.1.2 Clúster Beowulf

Quan es tracta de programació paral·lela, l'estructura per excel·lència és el *clúster* Beowulf. Un *clúster* Beowulf, tot i no ser un concepte estrictament definit, es tracta d'un seguit de computadors, sovint d'especificacions similars o idèntiques, connectats via una xarxa local i utilitzant software que els permet repartir les tasques a executar entre ells. Una de les principals característiques d'un *clúster* Beowulf és que l'usuari que executa el programa no arriba mai a veure els nodes que conformen el *clúster*, sinó que interactua amb un màster darrera del qual s'amaguen els altres nodes.

Segons s'explica a l'article *The Roots of Beowulf*, [9], el primer *clúster* Beowulf va aparèixer el 1994, va ser el resultat del projecte Beowulf de la The National Aeronautics and Space Administration (NASA) desenvolupat a Center of Excellence in Space Data and Information Sciences (CESDIS) i originat amb l'objectiu de construir un *clúster* per valorar la utilitat de la computació paral·lela en problemes de les ciències de l'espai. Tot i ja existir en aquella època diverses sol·lucions relatives a la computació en paral·lel, la comunitat d'Earth and Space Sciences (ESS) trobava que aquestes solucions no compartien del tot els seus objectius i per aquest motiu van optar per

dissenyar un *clúster* propi del qual en tinguessin un control total i que poguessin adaptar a les seves necessitats.

La idea de poder construir un *clúster* amb material econòmic i el qual un controla en la seva plenitud va captar l'atenció de la comunitat de computació paral·lela que en va estendre el seu ús, també impulsant així l'ús independent de *clústers*. El que en un primer moment va ser un sistema amb un objectiu concret per a un grup de persones, va créixer ràpidament fins a trobar-se estès arreu del món. Actualment els camps on més s'utilitzen els *clústers* Beowulf són els de la ciència i les matemàtiques, els quals presenten problemes molt complexos que requereixen d'un alta capacitat de computació. Tot i així també podem trobar *clústers* Beowulf utilitzats per moltes altres raons com per exemple millorar la disponibilitat de serveis o també per servir com a eina d'aprenentatge.

2.1.3 MPI i pas de missatges

Mpi és un estàndard que dicta com han de ser les llibreries de pas de missatges. Fixa la sintaxi de les diverses funcions i proposa un model únic de programa. Va sorgir a partir d'un consens entre més de 40 organitzacions les quals formen el fòrum d'mpi el 1993. El seu objectiu era idear un mecanisme de pas de missatges portable, flexible i eficient que es pogués utilitzar globalment.

El pas de missatges és un model en el món de la computació per el qual processos es comuniquen entre si mitjançant l'enviament de missatges per un canal qualsevol. D'aquesta manera dos processos poden compartir informació i fins i cooperar entre ells; tot això sense la necessitat que es trobin físicament en el mateix sistema.

En aquella època ja existia el model de pas de missatges, el qual moltes companyies havien adoptat per implementar en les seves arquitectures. Tot i això, encara no existia una manera estàndard de com implementar aquest model. Les diferents llibreries de cada companyia, tot i funcionar de forma molt semblant, presentaven petites diferències que en dificultaven la compatibilitat, sobretot si es volien connectar sistemes amb diferents arquitectures. Aquesta problemàtica va impulsar als diversos autors de llibreries a unir-se per formar el fòrum mpi.

La figura 2.2 mostra la gènesi de mpi i els seus objectius.

La primera definició de l'estàndard apareix el 1994 com a resultat de l'esforç comú del fòrum mpi. No va ser ni un any més tard que van començar a aparèixer les primeres implementacions aplicant l'estàndard. Amb aquestes implementacions, mpi es va anar estenent globalment fins a convertir-se en el que actualment és la norma pel que fa a aplicacions de pas de missatges.

El 1997 n'apareix la segona versió, mpi-2, que introdueix un gran ventall de correccions i extensions. Mpi-2 es centra en concretar com ha de ser el procediment de creació de processos, les comunicacions en un sentit i l'ús paral·lel de les interfícies d'entrada/sortida. El 2012, per acomodar les noves necessitats dels usuaris i estendre les funcionalitats de l'estàndard, aquest s'actualitza a la versió número tres (mpi-3). Entre moltes altres coses, aquesta actualització introdueix *bindings* per a Fortran i noves operacions en la comunicació. Actualment el fòrum està treballant en definir l'estàndard 4.0, algunes de les funcionalitats que s'estan debatent són les següents:

- Comunicació per *stream*
- Suport a la tolerància de fallides

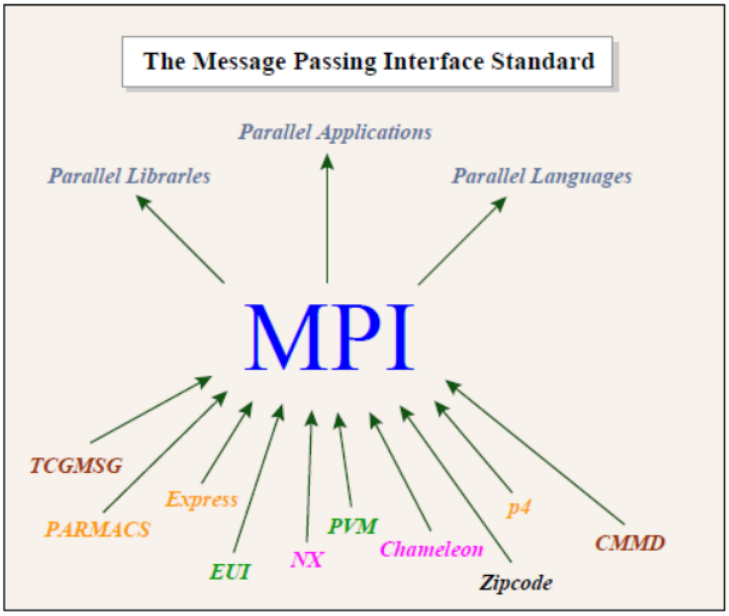


Figura 2.2: Esquema representatiu dels objectius que van motivar la creació d'MPI.

- Accés remot a memòria
- Millores de rendiment
- Operacions *large-count* (transferències d'elements de gran mida)

De forma molt semblant a PVM , MPI treballa amb el concepte de processos i no de màquines, de manera que es poden tenir diverses execucions paral·leles de l'aplicació sense necessitat que estiguin en diferents computadors. La distribució dels processos en els nodes del *clúster* ve a compte de l'usuari i es defineix abans de posar en marxa el programa. A més MPI és un estàndard de comunicació explícita, és a dir, en enviar un missatge s'ha d'indicar a quin procés va dirigit. Això provoca que els programes s'hagin d'escriure de forma determinada per què s'adaptin al conjunt de processos a utilitzar. Si es passa un missatge per exemple a un procés "7"però aquest no existeix en, el programa fallarà.

Actualment existeixen dues implementacions destacades de l'estàndard MPI. A continuació es comenten breument.

MPICH

MPICH és una implementació de l'estàndard de pas de missatges caracteritzada per un alt rendiment i per tenir una gran portabilitat. L'objectiu d'aquesta implementació és oferir un *framework* que serveixi tant per *clústers* petits i econòmics com per grans supercomputadors de manera que qualsevol pugui derivar-ne la seva implementació sense complicacions. MPICH va aparèixer el 1992 juntament amb l'estàndard MPI i ha crescut juntament amb ell. Existeixen versions de MPICH pera cada versió que ha anat apareixent de l'estàndard: Tenim MPICH-1 que s'adapta al primer estàndard, MPICH2 que es cenyeix a la segona versió i per últim la implementació actual, anomenada simplement MPICH , que s'adapta al tercer estàndard.

Actualment moltes empreses i universitats han utilitzat MPICH per contruïr les seves edicions de l'estàndard. Podem destacar IBM Platform MPI, Intel MPI, MVAPICH, entre moltes altres que podem trobar a la pàgina oficial de MPICH , [5]. Addicionalment, MPICH es troba present a 9 dels 10 supercomputadors més potents del món i l'any 2005 va guanyar el premi R&D100, [2], per la innovació tecnològica.

OpenMPI

Open Message Passing Interface (OpenMPI) és una implementació open source de l'estàndard MPI mantinguda per un grup de col·laboradors tant de la comunitat acadèmica i de recerca com de la industria. L'objectiu d'OpenMPI és el de, mitjançant el coneixement combinat de la seva comunitat, dissenyar la millor implementació lliure i gratuïta possible d'MPI.

OpenMPI té una llarga llista de contribuïdors que proveeixen de recursos al projecte i ajuden en el seu desenvolupament. En aquesta llista podem trobar grans organitzacions i companyies com són Amazon, Advanced Microdevices Inc. (AMD), IBM, Intel, etc. La llista complerta de col·laboradors es pot consultar a la pàgina oficial d'OpenMPI, [6].

2.2 OpenMP

En la programació paral·lela una altra tecnologia destacada és Open Multiprocessing (OpenMP). OpenMP ve a ser la tecnologia competidora amb MPI. Mentre MPI habilita el paral·lelisme en diferents hosts i amb memòria distribuïda, OpenMP es basa en el paral·lelisme de processos fent us de memòria compartida. De forma que un canvi en memòria fet per un procés serà visible a tots els altres de forma automàtica.

OpenMP ofereix una Application programming interface (API) molt senzilla d'utilitzar i un rendiment destacable, malauradament és una tecnologia no ideada per clústers ja funciona únicament per sistemes multiprocessador on els diferents nuclis tenen accés compartit a una mateixa memòria física. No ofereix paral·lelisme amb múltiples màquines independents.

El funcionament d'OpenMP es basa en fils (*threads*). Un programa OpenMP s'inicia com un únic procés que s'encarrega de generar un seguit de fils. Els fils comparteixen l'espai de memòria del seu procés pare i molts altres elements com són els *file descriptors* oberts, el directori de treball, les senyals, etc.

Pel que fa la història d'OpenMP, aquesta és bastant semblant a la de MPI. Durant la dècada dels 90 existien diferents fabricants de sistemes amb memòria compartida i tots ells oferien la seva pròpia solució per tal aprofitar-se d'aquest hardware.

Aquestes solucions, totes elles semblants entre si, es basaven en directives de fortran per senyalar trossos de codi que calia paral·lelitzar. Les directives de fortran són comentaris especials dins el codi que indiquen a l'ordinador que ha d'actuar de certa manera.

Tal i com diu Shirley Moore, [13], l'any 1994 va haver-hi un primer intent de dissenyar un estàndard per aquestes tecnologies. Tot i que d'aquest intent en van sorgir les bases d'ANSI X3H5, no es va aconseguir que l'estàndard s'adoptes. Durant aquests anys el processament distribuït es trobava en alça el que va ofuscar greument l'ús de la memòria compartida.

En els anys següents van aparèixer noves arquitectures de memòria compartida, cosa que en va reanimar el seu interès. Seguint aquest fet, l'any 1997 apareix l'estàndard OpenMP que continuava les idees deixades per X3H5.

OpenMP ha pogut seguir una evolució gradual i actualment es troba a la versió 4.5.

2.3 Supercomputadors i HPC

Un terme que ve ràpidament al cap en parlar de *clústers* són els supercomputadors, màquines grans i amb elevada capacitat de computació utilitzades per realitzar tasques complexes, en especial problemes científics i matemàtics que requereixen de tal potència. Un altre mot molt utilitzat i sinònim de supercomputació es el que en anglès es coneix com a High Performance Computing (HPC). Tot i que actualment pràcticament tots els supercomputadors es troben dissenyats en una estructura tipus *clúster*, en els seus orígens aquests eren dos conceptes separats.

El concepte de supercomputador precedeix al de clúster, els supercomputadors van aparèixer com a màquines amb la mateixa estructura que els ordinadors convencionals però amb diverses parts modificades i personalitzades per tal d'aconseguir un millor rendiment. El primer «esbós» de supercomputador del qual es té constància és el Control Data Corporation (CDC) 6600 que va sortir al mercat l'any 1964. Es tractava d'un ordinador d'un sol nucli que operava a 40MHZ i que es trobava optimitzat per a realitzat operacions matemàtiques.

Segons explica Gordon Bell en un dels seus articles, [4], el computador Cray-1 (1976) va ser el primer en ser pròpiament catalogat com a supercomputador. Cray1 utilitzava un processador de tipus vectorial, els quals es caracteritzen per que en executar una instrucció operen sobre un array de dades anomenades vectors, a diferència dels processadors normals que simplement actuen sobre un element per instrucció. A la figura 2.3 es pot veure un computador Cray en fase de desballestat.

Pel que fa a la computació paral·lela en els supercomputadors, aquesta ja va començar a introduir-se en aquest àmbit a finals de 60. Tot i això, els processadors vectorials van entrar en auge en els anys següents i màquines com el Cray-1 i les seves successores van dominar el sector durant dècades, oferint sempre el millor rendiment.

No va ser fins la dècada dels 90 que la supercomputació no va patir cap altre canvi considerable. Per llavors, la relació rendiment-preu dels processadors havia millorat considerablement respecte anys anteriors i de cop, el concepte de paral·lelisme va esdevenir viable. A partir d'aquest moment, la supercomputació va agafar un nou enfoc dirigit a la paral·lelització massiva que es va veure impulsat encara més gràcies a l'aparició del *clúster* Beowulf i MPI en els anys conseqüents.

És d'aquesta manera que els supercomputadors han evolucionat fins el que són a dia d'avui, on és impensable un supercomputador potent que no es tracti d'un *clúster* amb centenars o milers de nodes.

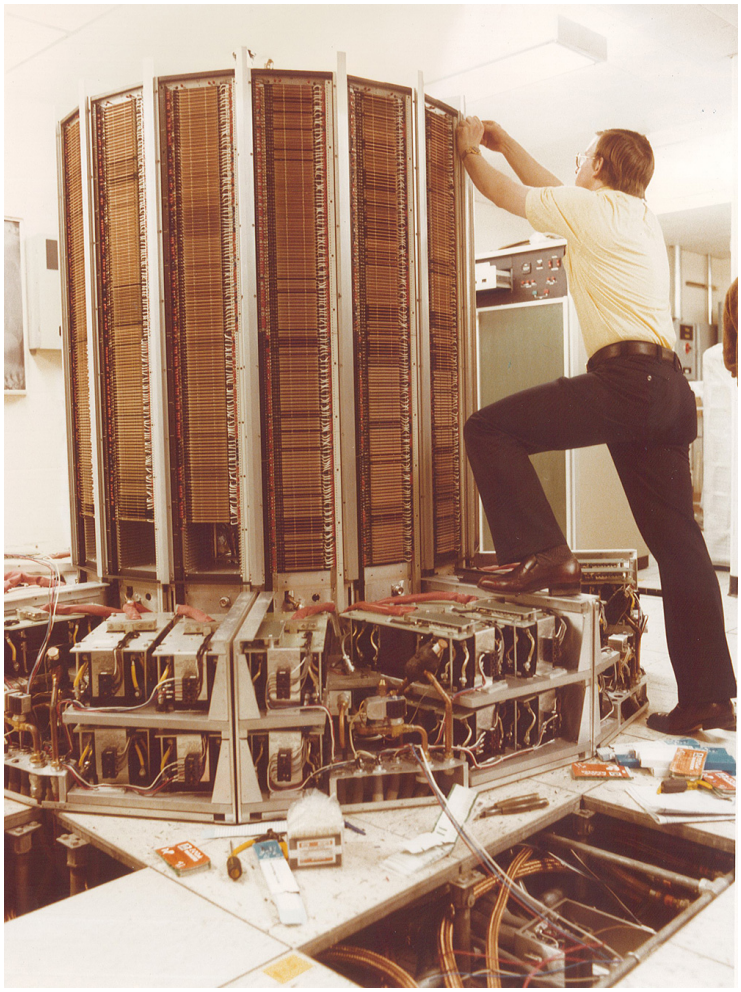


Figura 2.3: Supercomputador Cray-1, al NERSC, sent desmuntat.

Font: `https://www.extremetech.com`

2.4 Clústers en l'actualitat

2.4.1 Usos dels clústers

Actualment els *clústers* s'utilitzen extensivament i per objectius de tota mena. Tot i que fins aquest punt s'ha donat un enfoc principalment a la potència de càlcul, en realitat també són útils per altres propòsits. Segons aquests propòsits podem classificar els *clústers* en tres grans famílies.

- Clústers d'*alt rendiment*: Són aquells que busquen en el paral·lelisme obtenir una elevada capacitat computacional, per realitzar tasques pesades en el menor temps possible. En ells les tasques i programes es trossegen i es fa que cada node n'executi una part.
- Clústers d'*alta disponibilitat*: Són clústers que busquen oferir serveis amb un 100% de fiabilitat. El fet de tenir diverses màquines en paral·lel els ofereix una alta capacitat per corregir errors.. Per exemple, si un node falla sobtadament, el sistema pot desactivar

aquest node i delegar la feina que estava realitzant a un altre dels nodes. Quan això passa, el client que estava utilitzant el server no s'assebenta de res, per ell el servei funciona de forma normal.

- Clústers d'*alta eficiència/de carga balancejada*: Son *clústers* que tenen com objectiu oferir serveis amb resposta ràpida i a molts clients a la vegada. Per fer-ho reparteixen la feina de forma equitativa entre els diferents nodes. Tot i tenir un plantejament semblant, aquests clústers es diferencien amb els d'alt rendiment per el fet que en ells cada node executa una tasca independent, mentre què en els d'alt rendiment tots els nodes treballen en la mateixa tasca.

2.4.2 Exemples de clústers i HPCs

En aquest apartat s'investiguen diversos *clústers* existents per avaluar quines són les tecnologies més utilitzades i cap a on està evolucionant la ciència. Cal destacar que la facilitat per muntar un *clúster* actualment ha motivat la creació de milers d'ells, per el qual la tasca d'analitzar-los tots resulta impossible.

MareNostrum 3

MareNostrum és el nom amb que el Barcelona Supercomputing Center (BSC) anomena a les diferents versions del seu supercomputador. Fins ara MareNostrum ha estat sempre el supercomputador més potent de tota Espanya i actualment es troba en desenvolupament la seva quarta iteració. La figura 2.4 mostra una imatge d'aquest supercomputador.

El MareNostrum3, vigent des del 2012 fins el 2017, tenia les característiques que es mostren a la següent taula, amb les quals va aconseguir un rendiment de 1.1 Petaflops (10^{15} operacions de coma flotant per segon).

Arquitectura	Sistema operatiu	Paral·lelització	Nombre nodes
arm v8	Linux Suse 11	IntelMPI OpenMPI OpenMP	2056

Clúster Aiyara

El *clúster* Aiyara és un model de *clúster low-cost* i dissenyat el 2014 amb l'únic propòsit de processar Big Data. Es troba conformat per un conjunt de 22 plaques Cubiebord 1, ordinadors *on-board* i amb arquitectura ARM, semblants a las Raspberry Pi. Utilitza `ApacheHadoop` i `ApacheSparkm` uns programes dedicats al tractament de Big Data en sistemes distribuïts i que són molt populars. Tot i que no és típic utilitzar aquestes eines en arquitectures ARM, l'equip darrera el *clúster* Aiyara ha aconseguit processar un arxiu de 34GB en un temps raonable. A la figura 2.5 podem veure unes imatges del clúster aiyara.

Figura 2.4: Imatge del supercomputador MareNosturm, tal i com es veu a l'interior de la Torre Girona, Barcelona.

Font: `https://www.bsc.es`

2.4.3 Summit

D'acord amb [7], Summit, també conegut com OLCF-4, és un supercomputador desenvolupat per IBM per al Oak Ridge National Laboratory (ORNL), amb la col·laboració del fabricant de Graphics processing unit (GPU), NVIDIA. A data d'edició d'aquesta memòria, Summit és el supercomputador més potent del món. El seu motiu d'ésser és el d'utilitzar la supercomputació juntament amb la intel·ligencia artificial per buscar avanços en les ciències de la salud, de l'energia de partícules i en l'enginyeria.

Summit implementa les tecnologies típiques com són MPI, memòria compartida SHMEM entre moltes altres. El seu punt més peculiar, però, és que utilitza GPUs per accelerar els càlculs de coma flotant. Això permet al Summit realitzar fins a 2×10^{15} càlculs de coma flotant per segon amb un conjunt de 4608 nodes paral·lels. El seu sistema operatiu es tracta d'una implementació a mida de la distribució linux `redhat`.

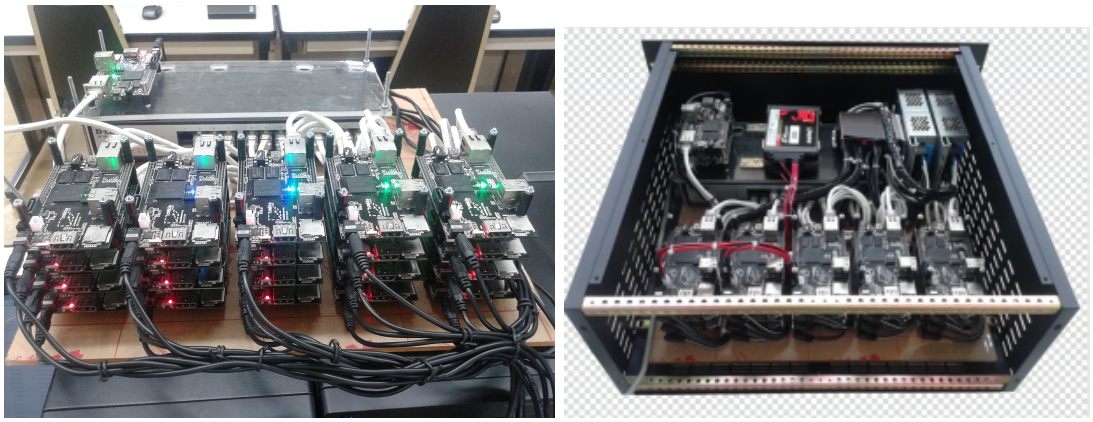


Figura 2.5: Prototip (esq.) i primera versió (dreta) del *clúster* Aiyara.

Font: `https://bananacluster.wordpress.com/tag/raspberry-pi`

Figura 2.6: Imatge on s'observen alguns dels *racks* que conformen el supercomputador Summit.

Font: `http://www.mellanox.com/blog/2017/11/what-does-it-mean-to-summit/`

3 Objectius

Havent observat l'estat de l'art de la programació paral·lela i els clúster, s'han fixat els següents objectius per el treball:

1. Dissenyar i implementar un *clúster* Beowulf amb suport per l'estàndard MPI. Per els computadors que el conformen s'ha triat utilitzar plaques Raspberry Pi ja que són un dispositiu altament disponible i econòmic, atributs que encaixen a la perfecció amb el concepte de *clúster* Beowulf.
2. Un cop muntant el clúster, executar en ell certs programes que facin ús de l'estàndard MPI per tal de validar el seu bon funcionament i també comprovar el rendiment del clúster.

Cal ser conscient de que, degut a la baixa capacitat de computació d'una raspberry, és possible que muntar un clúster d'elles no resulti rendible i que el preu no sigui competitiu considerant el rendiment final del *clúster*. Tot i això, en aquest treball el que es busca és aprendre sobre les tecnologies de clústers i computació paral·lela i el factor econòmic no és central.

4 Disseny del Sistema

En aquest capítol es descriuen tots els elements, tant de *hardware* com de *software* que s'han utilitzat en el nostre *clúster*. Per totes les aplicacions *software*, es fa una explicació de com funcionen i quines utilitats tenen.

És presenten `Octave`, `ipyparallel` i `Jupyter`, un seguit d'aplicacions molt utilitzades en la programació científica i que, en el nostre *clúster*, conformen el servei que s'ofereix als usuaris, de forma que amb elles poden escriure i executar programes paral·lels.

4.1 Hardware

El hardware utilitzat per construir el clúster el podem observar a la taula 4.1.

El clúster es troba format per 9 plaques Raspberry Pi (RPi), del model 3 B+ per ser concrets. Vuit d'elles són els nodes que s'encarreguen de fer la feina paral·lela mentre que la novena placa fa la funció de master. Totes elles es troben comunicades per una xarxa local.

A la figura 4.1 podem veure representada l'estructura esmentada, juntament amb altres elements que es comenten més endavant en l'apartat de configuració. La figura 7.6, mostra unes fotografies del muntatge.

4.1.1 Raspberry Pi 3 Model B+

RPi és un petit ordinador implementat en una sola placa i amb una mida aproximada a la d'una targeta de crèdit. Tot i tenir una capacitat computacional reduïda destaca per ser bastant barata, el que la fa una gran eina per l'aprenentatge. El nucli de les RPi utilitzades és un processador d'arquitectura ARM, de quatre nuclis i 1.4GHz. Degut a aquesta arquitectura poc

Nom	Unit	Cost/U(€)	Total(€)
Raspberry Pi 3 Model B+	9	33,99	305,91
SanDisk Ultra microSDHC UHS-I 16GB	9	7,00	63,00
Font Avance Technologies Inc. ATX-1125B 250W	1	10,00	10,00
Switch D-Link DES-1100-24	1	115,00	115,00
NanoCable 10.20.0101-BL	11	1,03	11,33
Sep. DREMEC TFM-M2.5X10/DR12	80	0,11	8,56
Sep. DREMEC TFF-M2.5X6/DR112	10	0,07	0,72
COST TOTAL			514.52

Taula 4.1: Material i cost del clúster

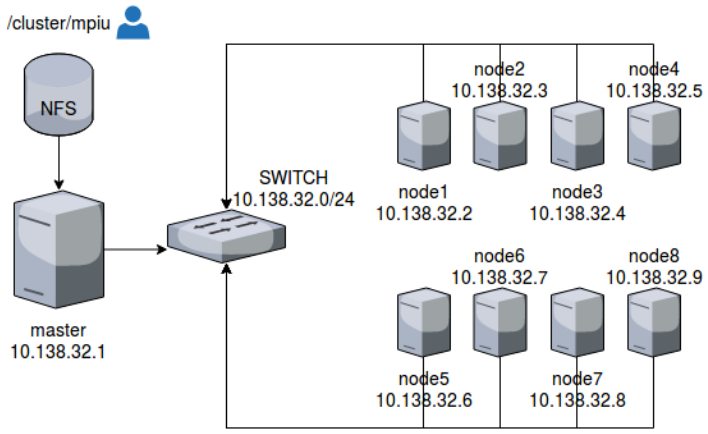


Figura 4.1: Esquema de xarxa del clúster.

Figura 4.2: Vista frontal i superior del clúster.

típica en ordinadors, el nombre de sistemes operatius que podem utilitzar és limitat. La fundació raspberry manté un seguit de distribucions i les ofereix a la seva pàgina web, [10]:

- Raspbian
- Ubuntu Mate
- Ubuntu Core
- RISC OS
- LibreElec

A part d'aquests també existeixen altres distribucions, mantingudes cadascuna per el seu corresponent creador. Alguns exemples són Arch linux ARM, CentOs, FreeBSD, Fedora ARM, etc.

A continuació es citen algunes de les especificacions més rellevants de la raspberry pi:

- Port Micro SD per emmagatzematge. Per si sola la RPi no incorpora cap sistema d'emmagatzematge, el que sí inclou és una ranura per una targeta Micro SD en la qual podem inserir el sistema operatiu juntament amb tots els nostres arxius i programes.

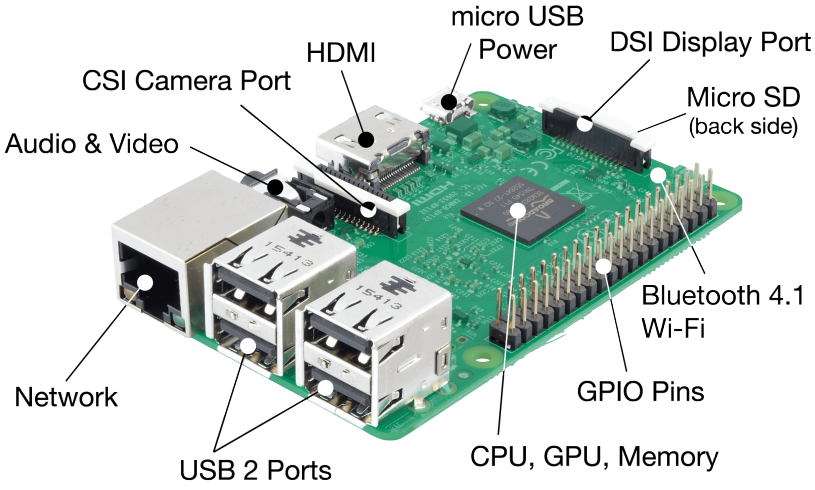


Figura 4.3: Raspberry Pi 3.

- La raspberry inclou tres sistemes de connectivitat: un port ethernet, un receptor de *Wi-fi* i un xip *Bluetooth*. Gràcies a això podem tenir una configuració més senzilla i ràpida ja que en tenir dues interfícies podem estar connectats al *clúster* i a la Internet sense haver de configurar *forwarding* i taules de ruta en el node master.

- 40 pins GPIO. Aquests pins d'ús genèric són totalment configurables per software. A més, un seguit de pins específics poden funcionar en els modes PWM, SPI, I2C i Serial el que facilita la integració de la Raspberry amb altres sistemes.

En la figura 4.3 podem observar una placa Raspberry Pi 3 i tot el conjunt d'elements que la conformen.

4.2 Sistema Operatiu

El sistema operatiu usat en el clúster és Raspbian. Raspbian és un sistema operatiu basat en Debian i especialment optimitzat per a funcionar en les RPi. Utilitzant aquest sistema operatiu es minimitzen el problemes específics de l'arquitectura ARM de RPi. Darrera d'aquest sistema operatiu hi ha una gran comunitat, fet que pot ser de gran ajuda si es presenta algun problema que no se sàpiga solucionar.

Un altre punt positiu de Raspbian és que, en estar basat en Debian, inclou la gran majoria de paquets necessaris per configurar el clúster en els seus repositoris. Malauradament Raspbian també té les seves desavantatges. Potser la principal és el fet que inclou bastants paquets molts dels quals segurament no utilitzarem, cosa que pot reduir l'espai del *clúster*. Algunes de les aplicacions que s'instal·len en aquest projecte requereixen bastant espai per ser compilades i es possible que calgui dedicar un esforç estona a purgar el sistema de tot el software que no faci servei.

4.3 MPI

Mpi és l'eina utilitzada per habilitar el processament paral·lel en el nostre *clúster*.

4.3.1 Funcionament

Prèviament a veure com funciona un programa mpi, cal definir alguns dels seus conceptes:

Comunicador Un comunicador és un objecte que defineix a un grup de processos que poden parlar enter ells.

Rank El rank és un enter únic que identifica a cada procés dins un comunicador. Utilitzant els ranks i els comunicadors, els processos poden enviar-se missatges de forma explicita, ja siguin missatges punt a punt o broadcasts.

COMM_WORLD Comunicador predefinit que engloba tots els processos paral·lels que s'han generat per el programa en el moment d'arrancada. A partir d'aquest comunicador se'n poden crear d'altres mitjançant funcions de divisió.

Tal i com s'observa a la figura 4.4, un programa mpi es pot separar en diverses etapes:

1. Importació de llibreries, entre elles la llibreria mpi.
2. Codi serie. Abans de començar a executar el codi mpi pròpiament el programa pot executar altres tasques.
3. Inici entorn mpi.
4. Codi paral·lel. Aquesta és la part del programa on els nodes poden utilitzar les funcions i mètodes descrits a la llibreria mpi per comunicar-se entre ells i col·laborar si escau.
5. Fi entorn mpi.

Tot i anomenar-se una part del programa «codi paral·lel», en realitat tota la execució del programa és paral·lela. En posar en marxa un programa mpi, aquest s'executa en un mateix moment a tots els nodes del *clúster*. No és, però, fins que s'engega l'entorn mpi que els nodes es poden veure entre ells i comença la «veritable» paral·lelitat.

4.4 Octave

GNU `Octave`, tal i com es descriu a [17], és tant un llenguatge d'alt nivell dissenyat per realitzar computacions matemàtiques, com un sistema interactiu amb el qual utilitzar aquest llenguatge. També pot ser utilitzat per realitzar tasques de càlcul de forma automatitzada.

El llenguatge Octave és un llenguatge interpretat i altament compatible amb Matlab, ja que comparteixen sintaxi. L'ènfasi del llenguatge es troba en les operacions amb matrius, per el qual disposa de multituds de. A part d'això, octave també permet moltes altres funcionalitats com per exemple creació de gràfics, programació orientada a objectes, crides a funcions de la llibreria estàndard de c, etc.

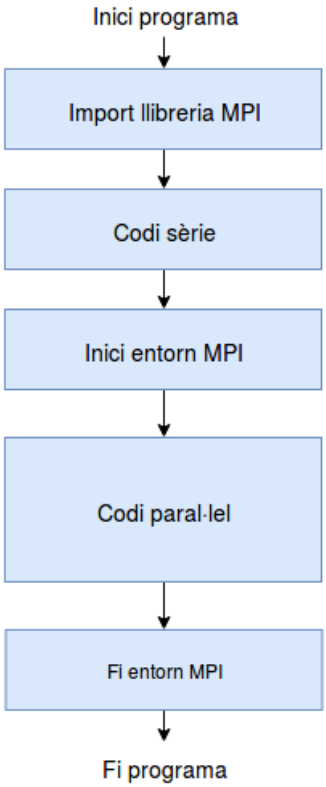


Figura 4.4: Etapes d'un programa MPI.

Un altre punt positiu és que les funcionalitats d'Octave es poden estendre fàcilment amb l'instal·lació de paquets addicionals. D'especial interès per nosaltres és el paquet MPI que afegeix a octave la capacitat d'executar codi paral·lel mitjançant l'estàndard de pas de missatges.

4.5 mpi4py

D'acord amb Lisandro Dalcin, [8], `mpi4py` és un mòdul python que adapta una implementació C++ del segon estàndard d'MPI al llenguatge de programació Python; de forma similar al paquet MPI per Octave.

Aquest mòdul ofereix una interfície orientada a objectes amb la qual podem comunicar processos de múltiples formes i, gràcies a la integració del mòdul `pickle`, ens permet serialitzar i enviar objectes python.

4.6 IPython: Jupyter i ipyparallel

Interactive Python (IPython), és un intèrpret d'ordres que afegeix funcionalitats extra respecte l'intèrpret estàndard de python. És una eina que es va crear buscant oferir una arquitectura rica

orientada a la computació interactiva i que ha tingut bastanta popularitat dins el món científic, ja que en resulta una bona eina de suport.

Dins el mateix projecte s'han creat també altres aplicacions que, utilitzant IPython com a *kernel*, adapten l'eina a altres propòsits o n'amplien les seves funcionalitats. Jupyter i ipyparallel formen part d'aquest conjunt de software nascut de IPython.

4.6.1 ipyparallel

IPython Parallel (IPYPARALLEL) és una branca de IPython que tracta amb la computació paral·lela. És compatible amb MPI, `mpi4py` i ens ofereix una capa d'abstracció; per la qual el software emmascara el clúster i ens ofereix un seguit de llibreries i funcionalitats per treballar amb ell.

La principal funcionalitat de ipyparallel és que aquest ens permet desenvolupar, executar i debugar tasques paral·leles de forma interactiva. També ens permet convertir el nostre clúster en un servei el qual poden utilitzar altres hosts remotament.

Tal i com se'ns explica al manual d'aquesta eina, [16], l'arquitectura de ipyparallel es troba separada en 4 parts o subsistemes que podem observar a la figura 4.5:

- El client
- Schedulers
- el HUB
- els motors

Motors

Els motors són sistemes que mitjançant el kernel de IPython executen codi com un servei de xarxa client-servidor, el qual escolta peticions d'execució de codi i les respon amb resultats corresponents.

Els motors són l'eina que habilita el paral·lelisme d' `ipyparallel`. Tenint un *clúster*, s'inicia un o més motors en cadascun dels nodes i llavors aquests es poden comunicar entre ells. Ja sigui mitjançant MPI o amb qualsevol altre dels mecanismes que ofereix el programa.

El HUB i els Schedulers

Els HUBs i els *schedulers* són dos subsistemes que conformen el que `ipyparallel` anomena el seu *controller*. La seva feina es gestionar els motors i la forma amb que es treballa amb ells. Alhora el *controller* és el sistema intermediari que connecta els clients amb els motors.

Els **schedulers** són uns processos que realitzen la feina que qualsevol esperaria tenint aquest nom. Es crea un scheduler per cada motor i de forma paral·lla als *schedulers* dels sistemes operatius, aquests habiliten l'execució concurrent de codi en els motors. D'aquesta forma dos o més clients poden utilitzar un mateix motor a la vegada amb una sensació de temps real.

El **hub** té la feina de gestionar les connexions de clients, els *schedulers* i tota comunicació entre clients i motors.

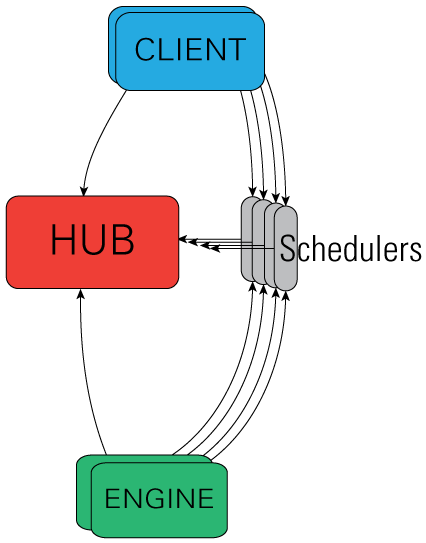


Figura 4.5: Esquema de l'arquitectura de ipyparallel.

Font: `https://ipyparallel.readthedocs.io`

El client

El client és un objecte mitjançant el qual un usuari pot interactuar amb un *clúster* de `ipyparallel` i enviar codi als seus motors. S'ofereixen dues formes d'interactuar amb el *clúster*, anomenades vistes.

Primer tenim la vista directa per la qual el client tria sobre quins i quants motors vol executar e seu codi. Per altra banda tenim la vista balancejada, per la qual el client executa codi sense especificar els motors; i el controller llavors intenta distribuir la feina de forma que tots els nodes treballin a una intensitat semblant.

4.6.2 Jupyter

El 2014 neix a partir de IPython el projecte `Jupyter`, per el qual es crea `Jupyter Notebook`. Aquest és un software que, entre moltes altres coses, converteix IPython en una aplicació web. `Jupyter Notebook` permet editar i compartir documents (*notebooks*) en línia amb la principal característica que en ells es pot inserir codi en viu, tipografies en `Tex`, imatges, equacions, etc.

Tot codi en un *notebook* es pot editar i executar de forma interactiva, a més s'han afegit nous llenguatges compatibles. Els principals que es poden utilitzar són `python`, `R` i `Julia`. A part d'aquests hi ha suport per a més de 40 addicionals, la majoria mantinguts per la comunitat.

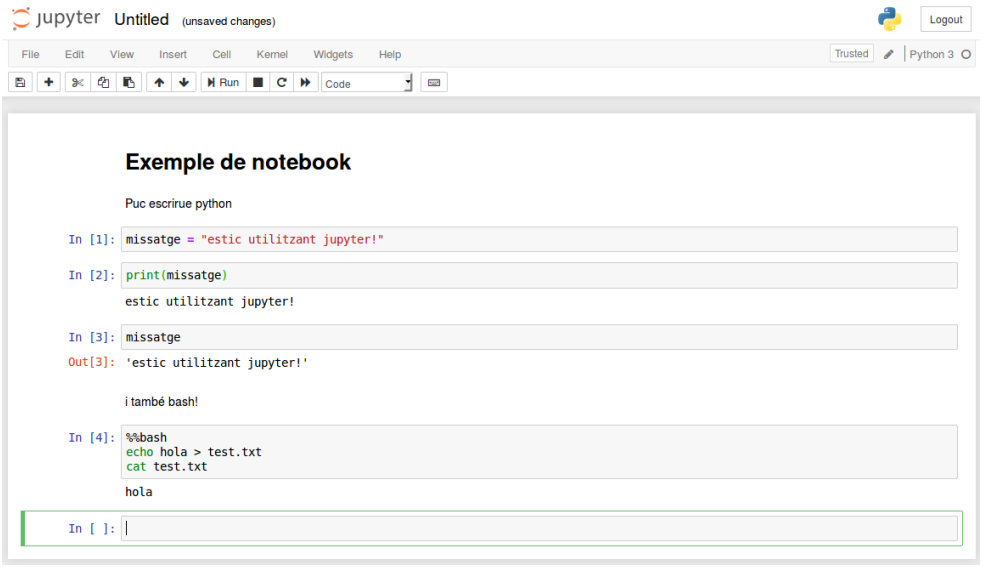


Figura 4.6: Notebook de Jupyter. S'hi pot observar codi python i bash.

A la figura 4.6 podem veure un exemple de *notebook* de `Jupyter`. Aquest es troba format per cel·les representades amb rectangles. En les cel·les podem escriure codi en diferents llenguatges, cosa que definirà la seva entrada. En funció d'aquesta entrada les cel·les també poden tenir una sortida, tal i com passa amb la cel·la 3 de l'exemple. A les sortides es mostra explícitament el resultat de la última comanda de la cell·la.

Tota cel·la executada queda marcada amb un enter que les ordena segons l'ordre en que s'han anant executant. A més les sortides romanen sempre visibles i en alguns casos se n'estilitza la seva aparença. Tot això junt permet a l'usuari comprendre fàcilment quina ha estat la línia d'execució del programa.

En poder executar codi python, `Jupyter` també ens serveix per establir una connexió amb un *clúster* de `ipyparallel`. A més, `Jupyter` es pot configurar com un servidor web típic, amb el qual es serveixen els *notebooks* a altres usuaris que poden estar en altres màquines. El codi en els *notebooks* s'executa en la màquina on hi ha el servidor. D'aquesta manera múltiples usuaris poden obrir o crear un notebook i posteriorment fer us del *clúster* com si d'un servei es tractés.

Cal afegir que `Jupyter` també ens permet engegar i parar clústers `ipyparallel` de forma senzilla des de la seva interfície web.

Jupyter dona punt i final al seguit de programari que s'ha instal.lat a sobre del nostre *clúster*. En la figura 4.7 és mostra un esquema per capes on es visualitza tot aquest muntatge, tal i com es troba configurat en les Raspberry Pi.

4.7 Virtualització

Gràcies al fet que un *clúster* Beowulf es troba conformat típicament amb màquines convencionals i amb sistema operatiu linux, podem virtualitzar-lo molt fàcilment.

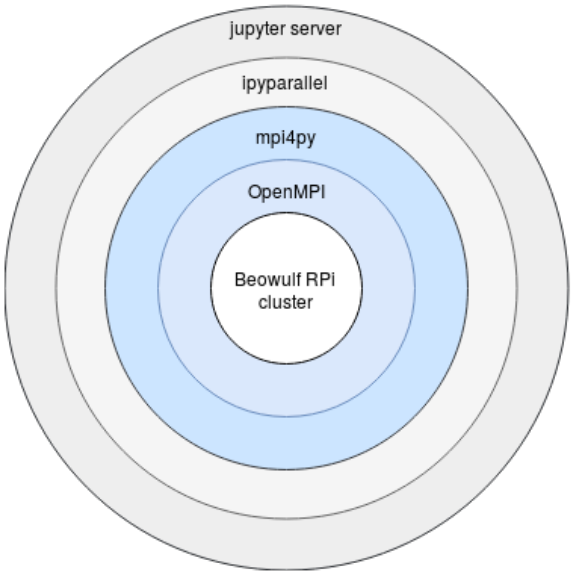


Figura 4.7: Esquema per capes del clúster amb el seu conjunt de programari.

Font: `https://ipyparallel.readthedocs.io`

El punt fort d'un *clúster* virtual és que configurar-lo resulta molt més senzill, ja que entre altres coses podem clonar màquines virtuals mentre que en un *clúster* físic s'ha'n de configurar els nodes un per un. Això ens facilita molt les cosses si simplement volem experimentar amb la programació paral·lela sense envolicar-se amb el *hardware*.

D'acord amb Kai Hwang, [12], la principal utilitat de virtualitzar *clústers* és que les maquines virtuals que tenim configurades es poden desplegar en cadascún dels nodes d'un *clúster* real. Això ens permet utilitzar les mateixes màquines virtuals com a nodes i obtenir paral·lelisme real, ja que efectivament s'estan executant en sistemes separats.

La virtualització també ens facilita la gestió del cluster. Per exemple, si es dona el cas que un node falla, aïllar-lo de la resta és tan fàcil com parar la seva màquina virtual.

En aquest projecte no s'ha arribat fer una implementació d'aquest tipus. Tot i així, s'és conscient de les seves avantatges i pot ser una bona millora de futur.

5 Configuració del sistema

En aquest apartat es descriu com, tenint prèviament el *clúster* ensamblat, podem preparar-lo per a executar programes MPI.

Instal·lar una implementació MPI i executar un programa és una tasca simple, tot i això, si el que volem es tenir diverses màquines cooperant hem de realitzar una configuració prèvia mitjançant altres eines.

5.1 Xarxa

El primer pas de tots és assegurar que els diferents nodes es coneguin entre ells. Els nodes es comunicaran constantment i per tant resulta necessari que coneguin les seves IPs per poder adreçar-se. Primer de tot assignarem una ip estàtica a cada node i després definirem les relacions nom-ip al fitxer `/etc/hosts`, el qual es el primer recurs que es consulta quan un ordinador amb linux vol conèixer la IP associada a un nom.

La via típica per establir una ip estàtica a la interfície ethernet d'una raspberry és editar el fitxer `/etc/network/interfaces`. Tot i així hem de tenir presents que Raspbian porta instal·lat predeterminada-ment el servei `dhcpcd`, el qual s'encarrega de configurar de forma dinàmica la nostra xarxa i per el qual qualsevol canvi que fem a interfaces serà ignorat. El que podem fer és indicar que volem una ip estàtica als fitxers de configuració de `dhcpcd`. Només cal afegir les següents línies a `dhcpcd.conf`:

```
...
interface eth0
static ip_address=10.138.32.2/24
```

Una alternativa seria desinstal·lar `dhcpcd` i fer la configuració a `interfaces`. Pel que fa l'arxiu `hosts`, aquest té el següent aspecte i es reparteix a tots els nodes que formen el *clúster*:

```
# IP          NOM
#======================
127.0.0.1       localhost
10.138.32.1     master
10.138.32.2     node1
10.138.32.3     node2
10.138.32.4     node3
10.138.32.5     node4
...
```

Per la configuració del *clúster* és necessari en algun moment que els nodes tinguin accés a Internet. Tot i que les raspberries utilitzades tenen una antena *Wi-fi* i se'ls hi pot configurar la interfície perquè es connectin a alguna xarxa; una altra alternativa viable és que el master faci de router i doni accés a la Internet als nodes (assumint que el master tingui accés a la xarxa). Per aquest cas cal modificar encara més el fitxer `dhcpcd.conf` dels nodes afegint la línia a continuació:

```
static routers=master
```

Això configurarà els nodes per tal que ennviin els paquest al master quan vulguin parlar amb alguna altra xarxa. Ara caldrà configurar master per tal que pugui enrutar paquets. Suposem que master té accés mitjançant la seva interfície `wlan0`.

```
# Per habilitar enrutament
mpiu@master~$ echo net.ipv4.ip_forward=1 >> /etc/sysctl.conf
# Per fer NAT
mpiu@master~$ sudo iptables -t a nat -A POSTROUTING -s 10.138.33.0/24 _
                -o wlan0 -j MAQUERADE
```

5.2 NFS i usuari per MPI

MPI requereix que el programa a executar sigui present a tots els nodes, aquest petit inconvenient pot ser fàcilment resolt amb un NFS. Network Filesystem (NFS) és un *filesystem* distribuït que es serveix per xarxa de manera que ordinadors clients poden muntar-lo i així obtenir accés remot a fitxers.

A part d'això, per executar el programa, MPI fa ús d'SSH utilitzant un usuari comú a tots els nodes. Haurem de configurar aquest usuari nosaltres mateixos a totes les raspberries i en això NFS ens pot ajudar a paral·lelitzar la seva configuració.

Per tenir funcionanet un NFS cal seguir els passos que s'indiquen a continuació.

5.2.1 Creació usuari del clúster

Podem crear un usuari nou amb la comanda `adduser`. Si no s'especifica el contrari, se li generarà un directori HOME amb el seu nom a `/home`.

```
user@node~$ adduser mpiu
```

Per comoditat i amb la mateixa comanda també podem afegir aquest nou al grup d'usuaris que poden utilitzar `sudo`:

```
user@node~$ adduser mpiu sudo
```

5.2.2 Configuració del host

El server de host d'NFS no ve inclòs amb Raspbian, només ve el servei de client. Podem obtenir-lo instal·lant el paquet `nfs-kernel-server`.

```
user@master~$ sudo apt-get install nfs-kernel-server
```

En el node master triem un directori que volem compartir. En el nostre cas ens interessarà compartir la configuració d'SSH que es troba al directori `.ssh` dins HOME. També podem crear un directori amb el qual compartir tot tipus d'arxius, entra altres els programes que volem executar amb MPI. Per especificar què volem compartir, editem el fitxer `/etc/exports` afegint-li les següents línies:

```
/home/mpiu/.ssh 10.138.33.0/24(rw,sync,no_root_squash,no_subtree_check)
/home/mpiu/cluster 10.138.33.0/24(rw,sync,no_root_squash,no_subtree_check)
```

Si alguns dels directoris que es volen compartir no existeixen, abans s'hauran de crear. A continuació reiniciem el procés servidor de NFS per a què s'apliquin els canvis.

```
user@master~$ sudo service nfs-kernel-server restart
```

5.2.3 Configuració dels clients

Ara per la banda dels clients, que seran tots els altres nodes del *clúster*, hem de realitzar la configuració necessària per què es muntin els directoris servits per el master.

Des d'un ordinador podem conèixer els directoris que serveix un altre si en coneixem la seva ip amb la següent ordre.

```
user@node~$ showmount -a <hostname>
```

Si provem la comanda amb master i tot ha anat bé, la instrucció ens hauria de confirmar que evidentment s'està servint `.ssh/` i `cluster/`. Per ara muntar el NFS hem d'editar el fitxer `/etc/fstab` en el qual s'especifiquen les particions o dispositius que s'han de muntar en el moment de *boot*. En ell afegim una nova línia on especifiquem el *filesystem* remot, el directori on el volem muntar i el tipus de muntatge (NFS).

```
master:/home/mpiu/cluster /home/mpiu/cluster nfs
master:/home/mpiu/.ssh    /home/mpiu/.ssh
```

Els canvis es realitzaran en reiniciar el sistema o en forçar el muntatge dels filesystems.

```
user@node~$ reboot
```

o bé

```
user@node~$ sudo mount -a
```

5.3 SSH

5.3.1 Funcionament

En executar un programa MPI el master requereix accedir a cada un dels nodes del clúster i executar en ells el programa remotament. Com ja s'ha dit aquest pas es realitza mitjanant Secure Shell (SSH). En el nostre cas utilitzarem el paquet `openssh-server` que es troba a la majoria de distribucions de linux.

SSH és una eina que ens permet accedir de manera segura a les terminals de *hosts* remots. Té una estructura típica de client-servidor i disposa d'un protocol d'autentificació que de manera predeterminada demana al client d'SSH la contrasenya de l'usuari al qual volem accedir. Com que existeixen també clients automatitzats com és el cas del nostre, SSH ofereix altres alternatives per autentificar-se. Una d'elles i la qual utilitzarem és l'autentificació per clau pública.

L'autentificació per clau pública espera que tant client com *host* tinguin generats un parell de claus pública i privada amb el quals xifraran la informació que volen enviar-se. Com amb tots els algoritmes de clau pública, la informació xifrada amb la clau pública només pot ser desxifrada amb la clau privada corresponent i a l'inrevés. En el moment que un client es vulgui connectar a un host aquest li enviarà la seva clau pública, si el host accepta la connexió llavors ell envia la seva. Un cop els dos tenen la clau pública de l'altre ja poden parlar. La condició per acceptar la connexió és que la clau pública del client es trobi a la llista de claus autoritzades en el host.

La manera més fàcil per configurar l'SSH en el nostre *clúster* és crear un mateix usuari amb la mateixa configuració en tots els nodes, de manera que amb el seu parell de claus es connecti a ell mateix però en altres màquines. L'existència del NFS que hem configurat prèviament ens facilita molt aquesta tasca. Com que compartim el directori `$home/.ssh` on s'emmagatzema la configuració d'SSH per mpiu, configurant l'usuari a un node ho farem a la vegada per a tots.

5.3.2 Configuració

Els passos per configurar tot el que s'ha esmentat són els següents:

1. La funcionalitat SSHja ve instal·lada a Raspbian però segurament la trobem deshabilitada. Podem habilitar el seu ús al menú de configuració de Raspbian dins *interfacing Options*, tal i com es veu a la següent imatge 5.1:

   ```
   user@node~$ sudo raspi-config
   ```

2. Generem el parell de claus per al nou usuari mpiu. En generar les claus triem la configuració predeterminada per què les claus es desin a $HOME/.ssh/.

   ```
   user@node~$ su mpiu              (canviem d'usuari a mpiu)
   mpiu@node~$ ssh-keygen -t rsa    (generem claus rsa)
   ```

3. Autoritzem la connexió del mateix usuari mpiu des d'altres nodes.

   ```
   mpiu@node~$ cd $HOME/.ssh
   mpiu@node~$ cat id_rsa.pub » authorized_keys
   ```

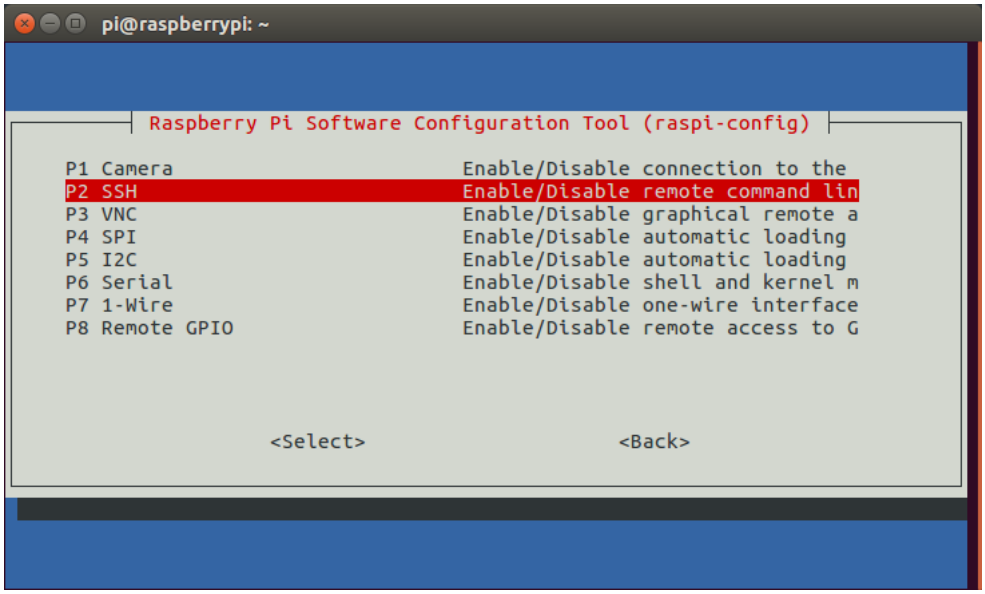


Figura 5.1: Menú de configuració per la raspberry pi (raspi-config).

5.3.3 Xifrat de la clau privada

En generar el parell de claus se'ns demanarà si volem xifrar la clau privada amb una clau simètrica per raons de seguretat. Si això es fa, mpi no podrà utilitzar-la de manera automàtica. Quan intenti utilitzar la clau privada se la trobarà xifrada i SSH li demanarà entrar la contrasenya per desxifrar-la. Podem sol·lucionar aquesta situació amb `keychain`.

`Keychain` és un servei que ens gestiona de manera segura les nostres claus GPG i SSH. Amb `keychain` només haurem d'escriure la contrasenya un sol cop en engegar la màquina i no cada cop que volem establir una comunicació.

Haurem d'instal·lar `keychain` a tots els nodes amb la següent comanda:

```
mpiu@node~$ sudo apt-get install keychain
```

Llavors editem el fitxer `.bashrc` al `home` de mpiu i afegim la següent línia:

```
eval ‘keychain --eval --agents ssh id_rsa‘
```

Amb aquesta línia `keychain` espera trobar el parell de claus al directori `.ssh` i amb nom `id_rsa` i `idv_rsa.pub`, és a dir, la configuració predefinida. Si ara tanquem la sessió de mpiu i tornem a entrar veurem que abans d'accedir a la *shell* se'ns demanarà la clau per desencriptar la clau privada d'SSH.

Amb això tindrem l'SSH totalment configurat i l'usuari mpiu podrà parlar amb ell mateix entre diferents màquines sense necessitat d'introduir cap clau d'accés.

5.4 MPI

L'ultim pas per deixar el *clúster* preparat és instal·lar MPI en sí. Tant MPICH com OpenMPI es troben als repositoris de debian però hi ha l'inconvenient que no es troben a la seva última versió i la instal·lació dels dos en un mateix directori pot portar problemes per ambigüitat de nom en alguns dels seus arxius. És par això que es recomana instal·lar-los a partir de source i cadascun dins de directoris aïllats. Per no complicar les coses en el projecte s'ha instal·lat únicament OpenMPI ja que ofereix algunes funcionalitats de més que en fan l'ús més còmode.

A part de la llibreria que implementa pròpiament l'estàndard, en instal·lar una implementació d'MPI també obtindrem un seguit d'executables:

- `mpiexec` i `mpirun`: Per una banda tenim els executors de codi paral·lel d'MPI. S'encarreguem de que el programa s'executi a diversos ordinadors i en múltiples processos, segons indiqui l'usuari. `Mpiexec` és el nom de l'executable segons marca l'estàndard. Tot i això algunes implementacions van començar amb mpirun i han triat seguir desenvolupant un executable amb aquest nom, el qual pot mostrar petites divergències respecte l'altre. Tot i això la forma de funcionar dels dos programes és exactament la mateixa.
- `mpicc`: *Wrapper* de `gcc` per compilar codi en C.
- `mpic++` i `mpicxx`: *Wrappers* de gcc per compilar codi en C++

5.4.1 OpenMPI

OpenMPI també es troba als repositoris, repartida en diferents paquets. Malauradament, aquesta versió es bastant antiga i ha portat problemes amb els programes que he volgut utilitzar, principalment octave. Com ja s'ha dit abans L'altre mètode que hi ha per instal·lar OpenMPI és partint dels arxius source. És més complicat però d'aquesta manera obtenim l'última versió estable. Podem obtenir els source a la pàgina d'OpenMPI.

Un cop descarregat i descomprimit el tar amb els arxius, podem compilar i instal·lar-los amb les comandes típiques `configure` i `make`. Per a RPi hem d'utilitzar uns *flags* especials a l'hora de la compilació o per contra saltarà un error queixant-se que el processador no és compatible.

```
mpiu@node~$ CCASFLAGS=-march=armv7-a CFLAGS=-march=armv7-a \
            ./configure --prefix=/usr/local/openmpi-3.0.1
mpiu@node~$ make
mpiu@node~$ sudo make install
```

Això ens instal·larà OpenMPI al directori `/usr/local/OpenMPI-3.0.1` i trobarem els diferents executables a `/usr/local/OpenMPI-3.0.1/bin`. D'aquesta forma tenim una instal·lació neta i aïllada fàcil de gestionar.

OpenMPI requereix que el directori `bin` amb el seus executables es trobin en la variable d'entorn PATH i el seu directori de llibreries es trobi a la variable LD_LIBRARY_PATH, això per tots els nodes. Podem modificar aquestes variables d'entorn del sistema de forma permanent i per a tots els usuaris si editem `/etc/bash.bashrc` afegint el següent parell de línies:

```
PATH="$PATH:/usr/local/openmpi-3.0.1/bin"
LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/local/openmpi-3.0.1/lib"
```

Cal escriure aquestes línies al principi del document, abans de la línia `[ -z "$PS1" ] && return`. Si això no es fa les línies seran ignorades per sessions d'ssh no interactives com és el cas d'MPI.

El principal efecte de modificar la variable PATH és que ara el sistema mirarà el directori especificat quan busqui algun executable. De forma que podrem cridar les comandes d'OpenMPI sense necessitat d'estar al directori on les hem instal·lat.

5.4.2 Comprovació

Un cop instal·lat MPI hauríem de tenir al nostre sistema els executables mencionats anteriorment.

- `mpiexec`
- `mpirun`
- `mpicc`
- ...

Per acabar i verificar que el *clúster* funciona podem executar un simple programa MPI. El següent codi C implementa un *hello world* en el què cada procés llegeix el seu identificador, el nom del node en què es troba i escriu un missatge per pantalla amb la informació recollida.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char **argv)
{
  int err, size, id, length;

  char name[MPI_MAX_PROCESSOR_NAME];
  //Iniciem l'entorn MPI
  err = MPI_Init(&argc, &argv);
  //Cada proces llegeix el seu id dins el comunicador principal
  err = MPI_Comm_rank(MPI_COMM_WORLD, &id);
  //Cada proces llegeix el nombre total de processos
  err = MPI_Comm_size(MPI_COMM_WORLD, &size);
  err = MPI_Get_processor_name(name, &length);
  //Cada proces treu per pantalla el missatge
  printf("%s: Hello World from process %i out of %i processes \n",
        name,id+1, size);
  //Finalitzem l'entorn MPI
  err = MPI_Finalize();
  exit(0);
}
```

Abans de poder executar el programa l'hem de tenir compilat. Podem utilitzar `gcc` però hauríem d'incloure diverses llibreries addicionals i configurar unes quantres variables d'entorn, cosa que ens faria aquesta feina un tant enrevessada. Per la nostra sort existeixen els *wrappers* de `gcc`

mencionats abans, que ens fa la feina tediosa per nosaltres. La seva forma d'ús és paral·lela a la de `gcc`:

```
mpiu@master~$ mpicc -o helloworld helloworld.c
```

Un altre prerequisit és haver-se connectat prèviament a cadascun dels nodes del *clúster* via SSH. Quan es tracta de la primera connexió realitzada a un host, SSH demana una confirmació de si de debò es vol connectar. MPI no sap respondre a aquesta situació i si això succeeix veurem que el programa es queda bloquejat sense fer res.

Havent confirmat els dos punts previs, podem passat a executar el programa amb mpiexec o mpirun. Hi ha diverses maneres de fer-ho, cadascuna més o menys convenient en funció de la complexitat en la distribució del programa en els nodes. Per el nostre cas ens serveix el següent:

```
mpiu@master~$ mpiexec -n 9 -host node1:4,node2:1,node3:1,node4:2 \
      ./helloworld
```

`hostfile` és un fitxer de text on s'especifiquen els *hosts* en els quals es vol executar el programa i els slots disponibles per a cadascun, dit d'altra forma, el nombre de processos paral·lels que es vol permetre executar per host. Aquest número, per obtenir una execució paral·lela real, sol ser el nombre de nuclis físics del processador del host en qüestió; tot i que se li pot assignar qualsevol altre valor.

Un arxiu hostfile té una estructura com la següent:

```
node1 slots=4
node2 slots=1
node3 slots=1
node4 slots=3
```

Executant la comanda vista abans amb l'anterior exemple de *hostfile* es creen 9 instàncies del programa *helloworld*: 4 en node1, 1 en node2, una altra en node3 i 3 en el quart node node4. El resultat que observaríem és el següent:

```
node1: Hello World from process 1 out of 9 processes
node1: Hello World from process 2 out of 9 processes
node1: Hello World from process 4 out of 9 processes
node2: Hello World from process 5 out of 9 processes
node1: Hello World from process 3 out of 9 processes
node4: Hello World from process 7 out of 9 processes
node3: Hello World from process 6 out of 9 processes
node4: Hello World from process 8 out of 9 processes
node4: Hello World from process 9 out of 9 processes
```

Si rebem això significa que tenim un *clúster* amb MPI funcional. Cal dir que l'ordre en que es reben els missatges pot no ser el mateix ja que l'ordre i velocitat d'execució dels programes en els nodes és no determinista.

5.5 Creació d'imatges de disc personalitzades

El procés de configuració pot ser tediós i durador, sobretot si s'ha de realitzar en un llarg seguit de nodes. Una tàctica molt útil per aquests casos és un cop tenim una primera raspberry configurada, fer una copia del seu *filesystem* per inserir-la a les altres plaques. El fet que les particions amb les dades es trobin a una targeta SD, que generalment és de tamany reduït, no fa més que facilitar-nos aquesta tasca. Per aquest procediment podem utilitzar la instrucció `dd` seguint els següents passos:

1. Traiem la targeta micro-SD de la nostra RPi i la inserim a un altre ordinador amb linux.
2. Executem la següent instrucció per llistar els dispositius connectats al nostre ordinador:

   ```
   usr@pcr~$ df -h
   ```

   Veurem que apareixen dos dispositius nous que abans no hi eren, aquests corresponent a les dues particions que es creen en instal·lar Raspbian. Els dos dispositiu solen tenir un nom com `/dev/mmcblk0p1` i `/dev/mmcblk0p2` en els quals `p1` i `p2` identifica quina partició és cadascuna i `/dev/mmcblk0` identifica la targeta on es troben.
3. Per generar la copia de la targeta fem el següent:

   ```
   usr@pcr~$ sudo dd bs=4M if=/dev/mmcblk0 of=myraspbian.img
   ```

   Això ens generarà una imatge de disc `.img` amb el nom `myraspbian` i exactament els mateixos continguts que hi ha a la micro-SD.
4. Per últim podem instal·lar aquesta imatge a una altra targeta de la mateixa manera que hem instal·lat Raspbian prèviament.

6 Configuració de les Aplicacions

6.1 GNU Octave

`Octave` ha estat el programa que més problemes ha donat. Principalment perquè la versió actual d'octave als repositoris i el paquet `mpi` tant d'`Octave` forge com dels repositoris de debian no funcionen correctament junts. Resulta que el paquet mpi ja no es manté en aquests dos llocs i no hi cap tipus de senyalització que ho indiqui.

S'ha trobat una combinació funcional en `octave-4.4.0` (la última versió en el moment de realitzar el treball) juntament amb el paquet `mpi` en la versió 2.2.0, que s'ha pogut trobar afortunadament en un fòrum en línia on participava la persona encarregada de mantenir el paquet.

Podem obtenir `Octave-4.4.0` instal·lant-lo des del source. Per fer-ho cal haver afegit prèviament el directori d'instal·lació d'OpenMPI a la variable PATH. També es necessari instal·lar abans totes les dependències del programa, les quals per sort podem obtenir de la versió actual d'octave als respositoris. L'única dependència diferent entre les dues versions és `Qt`, per la interfície gràfica. Octave-4.4.0 utilitza la versió 5 de Qt mentre que la versió 4.2.0 dels repositoris utilitza `Qt4`. Afortunadament tenim l'alternativa de compilar `Octave-4.4.0` amb una versió anterior de Qt així que aquest problema no ens ha d'amoïnar massa.

Per poder instal·lar només les dependències cal configurar `apt`. Descomentat la corresponent línia `dev-src` a `\etc\apt\sources.list` podem permetre la descàrrega de fitxers source. Llavors podem obtenir Octave amb el següent seguit de comandes:

```
% Instal·lem dependències
mpiu@node~$ sudo apt-get update
mpiu@node~$ sudo apt-get source octave
% Instal·lem Octave del source
mpiu@node~$ cd octave-4.4.0
mpiu@node~$ ./configure --with-qt=4 --prefix=/usr/local/octave-4.4.0
mpiu@node~$ make
mpiu@node~$ sudo make install
```

6.1.1 Octave MPI

El paquet MPI d'octave també el trobem als repositoris de debian, el que ens facilita la feina.

De moment aquest paquet es limita a adaptar les funcions bàsiques d'MPI per tal d'habilitar la computació paral·lela en Octave. Ens ofereix les típiques funcions d'obrir l'entorn MPI, enviar i rebre missatges, etc; de manera que les podem inserir en els nostres scripts .m d'octave i amb elles construir el nostre programa seguint les guies de l'estàndard. A dia d'avui només hi ha

suport per l'execució dels scripts via bash, així que no podem utilitzar MPI de forma interactiva amb l'intèrpret d'octave.

El paquet s'ha d'instal·lar des de dins octave mitjançant la comanda següent. És necessari, però, tenir l'executable mpicc d'OpenMPI a la variable PATH. Si ja ho hem fet abans llavors ja no hi haurà cap problema.

```
octave> pkg install mpi-2.2.0.tar.gz
```

Arribat aquest punt ja podem utilitzar MPI amb octave. Per executar un programa podem fer servir les instruccions mpiexec i mpirun com es mostra a la comanda següent, que calcula Pi amb un dels programes de mostra que incorpora el paquet MPI.

```
# Desde bash
mpirun -n 2 -H node1:2 octave -q --eval 'pkg load mpi; Pi() -")'
# Des d'octave
system("mpirun -n 2 -H node1:2  octave -q --eval 'pkg load mpi; Pi() '")
```

6.2 mpi4py

Podem realitzar la instal·lació de `mpi4py` amb la eina `pip` de python. Tot i això, per pode fer-ho cal modificar una variable del sistema temporalment per indicar el directori on es troba el nostre executable `mpicc` d'OpenMPI. Si aquest pas no es fa `pip` buscaria `mpicc` a `\usr\bin`, `\usr\local\bin` o `\bin` i evidentment no el trobaria, fallant així la instal·lació.

Si no es disposa de la eina `pip`, podem baixar-la via `apt`. Per tot el codi python d'aquest projecte s'usa python3, per tant baixarem pip3.

```
mpiu@node~$ sudo apt-get install python3-pip
```

Tenint `pip` podem obtenir `mpi4py` mitjançant la següent instrucció:

```
mpiu@node~$ env MPICC=/usr/local/openmpi-3.0.1/bin/mpicc/ pip3 install mpi4py
```

Això instal·larà mpi4pi exclusivament per l'usuari `mpiu` qui ja hem configurat per ser qui utilitzi el clúster. Si es volgués podem instal·lar mpi4py per tots els usuaris afegint sudo a la comanda. Independentment de quina opció es trii, amb això tenim mpi4py instal·lat i podem utilitzar-lo amb python important la llibreria amb el mateix nom.

6.3 ipyparallel

6.3.1 Configuració

`Ipyparallel` també pot ser instal·lat fàcilment mitjançant `pip`. La seva instal·lació és necessària a tots els nodes.

```
mpiu@node~$ pip3 install ipyparallel
```

Amb això se'ns instal·laran un seguit d'executables a `\home\mpiu\.local\bin`. Si els volem executar des de qualsevol lloc podem fer com amb OpenMPI i afegir aquest directori a PATH.

Per fer servir `ipyparallel` hem d'engegar tota la seva estructura de controlador i motors. Tot i que existeixen comandes per fer aquestes accions per separat, molt sovint és millor alternativa utilitzar la comanda `ipcluster` que engega tot el conjunt a la vegada. Aquesta comanda demana el nombre de processos paral·lls que volem i si no especifiquem res més el que farà es iniciar un clúster dins el host on hem executat la comanda, amb un nombre de motors igual als processos que hem demanat. A part d'això els motors no poden comunicar-se directament de forma predeterminada, la única forma que tenen per fer-ho és que el client reculli dades d'un motor i sigui ell qui les envii a un altre. Evidentment, aquest panorama no ens deixarà satisfets.

Per engegar `ipyparallel` sobre un *clúster* de màquines físiques, amb com a mínim un motor a cadascuna d'elles i que es puguin comunicar, el que hem de fer es generar un perfil de clúster (un conjunt de fitxers), configurar-lo adientment i dir-li a `ipyparallel` que volem utilitzar tal perfil.

Podem crear un perfil de *clúster* per `ipyparallel` amb la següent comanda.

```
mpiu@master~$ ipython3 profile create --parallel --profile=cluster_rpi_mpi
```

Això ens crearà un perfil amb nom «clúster_rpi_mpi» i generarà un seguit de fitxers a `~\.ipython\profile\_clsuter\_rpi\_mpi`. per tal de configurar-lo.

Els motors que es trobaran en altres màquines necessiten també poder veure el perfil i la configuració de ipyparallel. Cal afegir al NFS el directori `~\.ipython`, així de nou ens estalviem repetir la mateixa configuració per cada node.

Per ara dur a terme la configuració tal i com volem ens es suficient amb modificar el fitxer `ipcluster_config.py` dins la carpeta del perfil. Caldrà escriure les següents línies, cadascuna per un propòsit diferent:

```
%per processos de control
master slots=1
%per processos de treball
node1 slots=1
node2 slots=1
node3 slots=1
node4 slots=1
node5 slots=1
node6 slots=1
node7 slots=1
node8 slots=1
node9 slots=1
```

Figura 6.1: `hostfile` del clúster per ipyparallel.

Per configurar el controller

- `c.IPClusterEngines.engine_launcher_class = 'MPI'`

  Això iniciarà tots els processos motors dins un mateix entorn MPI amb mpiexec. De forma que podran executar codi MPI i enviar-se missatges entre ells.

- `c.IPClusterStart.controller_ip = '10.138.33.1'`

  Especifiquem una IP a la qual el controlador estarà escoltant perquè els motors el puguin trobar. Predefinida-ment aquest camp és localhost i per tant només s'escolten motors dins la mateixa màquina on hi ha el controlador. Per al nostre cas tindrem motors a cadascuna de les raspberry Pis del clúster i cal permetre que aquests puguin parlar amb el controlador. La IP hauria de ser la del master, on engegarem el controlador. Si els motors i el controller es troben en una xarxa local, com és en el nostre cas, també serveix posar '*' de IP.

- `c.IPCClusterEngines.daemonize = True`

  Opcional. Provoca que ipyparallel és comporti com un dimoni. En engegar ipcluster, la seva execució es realitzarà en segon pla en lloc de bloquejar una terminal.

Per configurar MPI

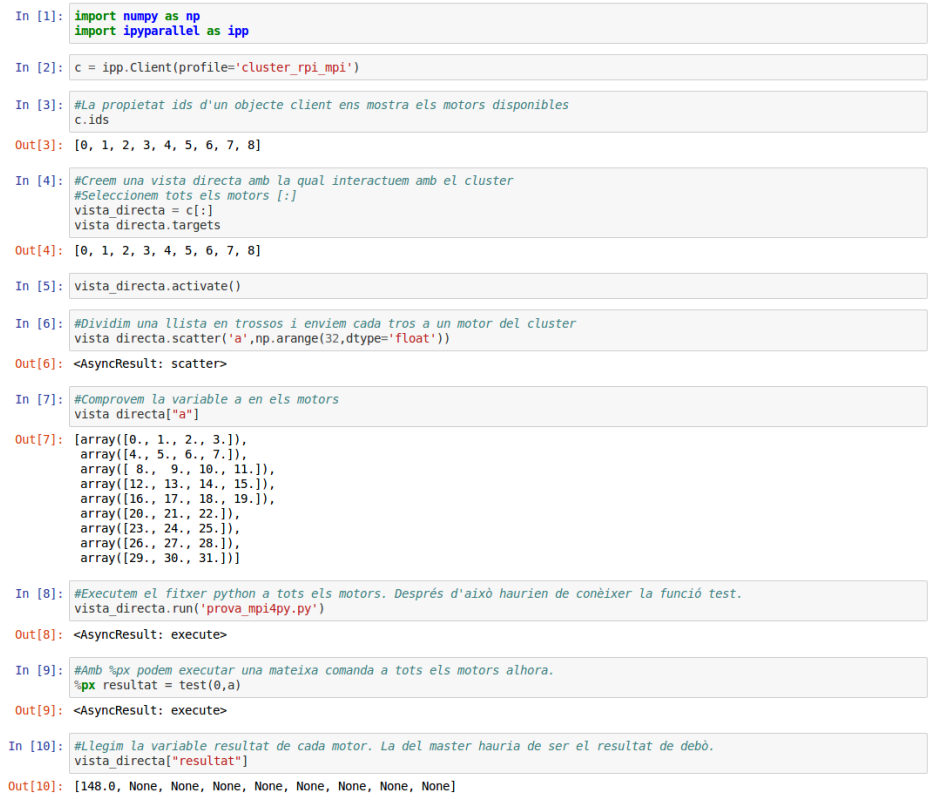
- `c.MPILauncher.mpi_args = [ ""-hostfile","/home/mpiu/cluster/hostfile"]`

  Indiquem que per generar l'entorn mpi utilitzi un hostfile determinat. Aquest hostfile hauria de ser com el que es pot veure a la figura 6.1, amb un slot mínim per cada integrant del clúster. Després el client via ipyparallel ja podrà triar sobre quants motors vol treballar però és imprescindible especificar-los tots al hostfile. Si en falta algun, no es crearà un procés motor en ell i per tant no serà accessible després.

- `c.MPI.use 'mpi4py'`

  Especifiquem que per la comunicació MPI s'utilitzi la llibreria d'`mpi4py`.

Ara ja configurat el perfil podem engegar un *clúster* que l'utilitzi amb la comanda `ipcluster`:



```
In [1]: import numpy as np
        import ipyparallel as ipp

In [2]: c = ipp.Client(profile='cluster_rpi_mpi')

In [3]: #La propietat ids d'un objecte client ens mostra els motors disponibles
        c.ids
Out[3]: [0, 1, 2, 3, 4, 5, 6, 7, 8]

In [4]: #Creem una vista directa amb la qual interactuem amb el cluster
        #Seleccionem tots els motors [:]
        vista_directa = c[:]
        vista_directa.targets
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8]

In [5]: vista_directa.activate()

In [6]: #Dividim una llista en trossos i enviem cada tros a un motor del cluster
        vista_directa.scatter('a',np.arange(32,dtype='float'))
Out[6]: <AsyncResult: scatter>

In [7]: #Comprovem la variable a en els motors
        vista_directa["a"]
Out[7]: [array([0., 1., 2., 3.]),
         array([4., 5., 6., 7.]),
         array([ 8.,  9., 10., 11.]),
         array([12., 13., 14., 15.]),
         array([16., 17., 18., 19.]),
         array([20., 21., 22.]),
         array([23., 24., 25.]),
         array([26., 27., 28.]),
         array([29., 30., 31.])]

In [8]: #Executem el fitxer python a tots els motors. Després d'això haurien de conèixer la funció test.
        vista_directa.run('prova_mpi4py.py')
Out[8]: <AsyncResult: execute>

In [9]: #Amb %px podem executar una mateixa comanda a tots els motors alhora.
        %px resultat = test(0,a)
Out[9]: <AsyncResult: execute>

In [10]: #Llegim la variable resultat de cada motor. La del master hauria de ser el resultat de debò.
         vista_directa["resultat"]
Out[10]: [148.0, None, None, None, None, None, None, None, None]
```

Figura 6.2: Exemple d'execució de codi paral·lel amb ipyparallel.

```
mpiu@master~$ \home\mpiu\.local\bin\ipcluster start -n 9 \
                --profile=cluster\_rpi\_mpi
```

Si hem activat la opció de dimoni, el sistema d'`ipyparallel` es posarà a executar en segon pla, si aquest no és el cas, veurem que el procés romandrà executant-se en el terminal en el qual l'hem invocat. Per parar el dimoni cal utilitzar `ipcluster` de nou:

```
mpiu@master~$ \home\mpiu\.local\bin\ipcluster stop \
                --profile=cluster\_rpi\_mpi
```

6.3.2 Utilització

Finalment, podem utilitzar el *clúster* via python amb l'objecte `Client`. En el següent exemple 6.2 es mostra com utilitzant una vista directa s'executa un petit script en els nodes del *clúster*.

En el programa es reparteixen unes llistes en els motors. Llavors és crida una funció `mpi4py` amb la qual cada motor calcula la mitjana dels valors de la seva llista i l'envia al master. El master llavors calcula la suma dels valors rebuts.

L'script python és el següent:

```python
#!/usr/bin/env python3
from mpi4py import MPI

def test(master,llista):
    comm = MPI.COMM_WORLD
    rank = comm.rank
    mitjana = 0
    if rank != master:
        mitjana = sum(llista)/len(llista)
    # Reduce realitza unn operacio amb una variable que s'ha de trobar a tots
    # els nodes i envia el resultat a un node desti (root)
    suma = comm.reduce(mitjana,op=MPI.SUM,root=master)
    if rank == master:
        print("La suma es %f" % suma)
        return suma
    return
```

6.4 Jupyter

`Jupyter` és una altre programa que pot ser instal·lat amb `pip`. La seva instal·lació es només necessària al node master.

```
mpiu@master~$ pip3 install jupyter
```

Tal i com ve instal·lat, Jupyter engega un servidor de notebooks al qual només s'hi pot accedir des de localhost. si volem convertir-lo en un servidor púbic cal realitzar una curta configuració.

Jupyter dona accés a la terminal de bash a tots els usuaris que es troben connectats, de forma que poden modificar molts arxius del sistema via comandes de la *shell*. El mateix es pot fer també dins les cel·les d'un *notebook*. És molt recomanable per aquest motiu limitar l'accés al servidor amb una contrasenya. Per afegir-ne una cal executar la següent comanda:

```
mpiu@master~$ \home\mpiu\.local\bin\jupyter notebook password
```

Ara, per tal de habilitat l'accés a altres màquines al servidor cal generar els arxius de configuració de Jupyter, que posteriorment editarem al nostre gust.

```
mpiu@master~$ \home\mpiu\.local\bin\jupyter notebook --generate-config
mpiu@master~$ cd $HOME/.jupyter/
mpiu@master~$ ls
jupyter_notebook_config.json
jupyter_notebook_config.py
```

L'arxiu a editar és `Jupyter_notebook_config.py`. En ell s'han d'afegir principalment les següents línies:

- `c.NotebookApp.ip = '10.138.33.1'`

  Aquesta línia especifica la ip a la qual es trobarà escoltant el servidor de Jupyter.

- `c.NotebookApp.notebook_dir = '/home/mpiu/cluster/jupyter'`

  Configura el directori de treball inicial del servidor. Tot usuari que es connecti al servidor podrà veure el contingut d'aquest directori, per tal és convenient canviar-lo a un directori de la nostra tria.

En ser d'un mateix creador, `ipyparallel` i `Jupyter` ofereixen certes funcionalitats addicionals en treballar junts. Una que pot ser d'interès és la pestanya de *clústers* de `Jupyter`, que es pot veure en la figura 6.3. En aquesta pestanya se'ns mostra una llista de tots els perfils de *clústers* disponibles en la màquina servidora. La pestanya també permet que qualsevol usuari connectat pugui engegar i parar *clústers* de forma senzilla simplement prement un botó. A questa utilitat ve desactivada de sèrie. Per habilitar-la em de recòrrer altra vegada a la comanda `ipcluster`:

```
mpiu@master~$ ipcluster nbextension enable
```

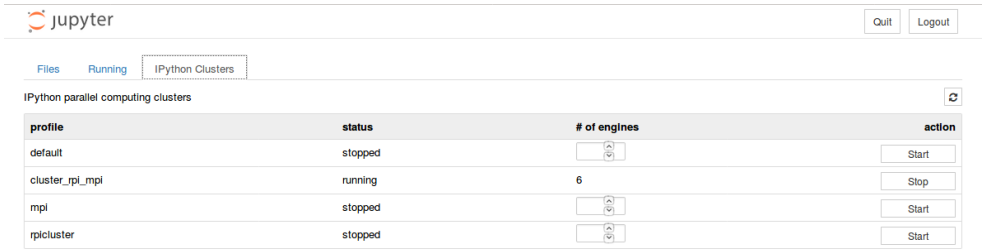


Figura 6.3: Pestanya per la gestió de clústers ipyparallel en la interfície de Jupyter.

Arribats a aquest punt, tenim `Jupyter` totalment configurat per el que ens interessa. Podem executar el programa amb la següent comanda:

```
mpiu@master~$ \home\mpiu\.local\bin\jupyter notebook password
```

Veurem com s'engega el servidor. Podrem connectarnos a ell amb la url `master~:8888`.

7 Tests i Resultats

En aquest capítol s'expliquen les proves que s'han fet sobre el *clúster* i la corresponent pila de software que s'ha configurat prèviament. L'objectiu d'aquestes proves és obtenir mesures que permetin tant caracteritzar el rendiment del *clúster* com validar i valorar el funcionament de MPI.

Les tres proves provenen de l'àmbit del càlcul numèric. La primera consisteix en un producte de matrius implementat sobre `Python`, la segona és el càlcul de π emprant el mètode de Monte Carlo sobre `Jupyter` i la tercera prova consisteix a calcular π amb integració numèrica sobre `Octave`.

Les proves són algoritmes simples i fàcilment paral·lelitzables. S'han experimentat sobre tecnologies diferents per il·lustrar l'ús de diverses eines. Finalment, els resultats de rediment s'han posat en relació amb el rendiment d'un computador de sobretaula i s'ha estudiat si els resultats empírics segueixen la llei d'Amdahl.

7.1 Multiplicació de matrius amb Python

L'objectiu d'aquest test és determinar el temps d'execució requerit pel *clúster* per a multiplicar dues matrius quadrades $A * B = C$. Aquesta tasca es paral·lelitza de forma que cada node esclau calcula el producte d'una submatriu de C i, un cop ha acabat, envia els seus resultats al procés mestre. El procés mestre s'encarrega de repartir la feina entre els nodes i posteriorment esperar la resposta d'ells per reconstruir la matriu C.

Per fer aquest càlcul s'ha utilitzat un script `Python` que usa els mòduls `mpi4py` i `matrixMultiplication.py` (vegeu l'annex 1).

L'experiment s'organitza de la següent forma. Es fan tests sobre matrius $n \times n$ per valors de $n = 24, 48.96, 192, 240, 384, 432$. Per cada dimensió de la matriu el test es fa 10 vegades amb la finalitat de disminuir la variabilitat de l'experiment. Cada test es fa sobre un *clúster* k nodes amb $k = 1, 2, 4, 6, 8$.

L'experiment es llença amb *script* de *bash* i els resultats de les múltiples execucions s'emmagatzemen en diferents fitxers de text. L'*script* és el següent:

```
#!/bin/bash
for STEPS in 24 48 96 192 240 384 432; do
    for TEST in 1 2 3 4 5 6 7 8 9 10; do
        for CPU in 1 2 4 6 8; do
          echo "Realitzant intent" $TEST "per" $CPU "nodes i matrius de dimensio" $STEPS
          mpiexec -hostfile hostfile -n $(($CPU + 1)) python3 _
          matrixMultiplication.py $STEPS $STEPS >> mpi$CPU.$STEPS.log
        done
    done
done
```

	$\bar{t}_{n,k}$ (s)						
k	$n = 24$	$n = 48$	$n = 96$	$n = 192$	$n = 240$	$n = 384$	$n = 432$
1	0,090	0,643	4,945	39,030	76,245	315,008	444,928
2	0,048	0,327	2,491	19,416	38,665	159,705	224,238
4	0,028	0,170	1,268	9,941	19,333	79,688	113,230
6	0,021	0,118	0,866	6,692	13,110	53,413	75,506
8	0,025	0,097	0,701	5,021	9,740	40,471	57,359

Taula 7.1: Temps d'execució del producte de matrius en el clúster

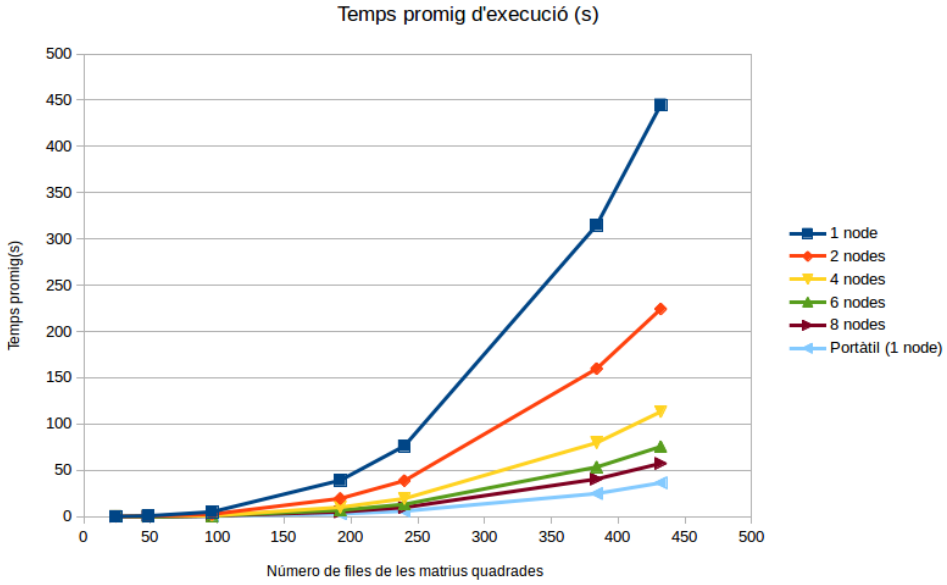


Figura 7.1: Mitjanes del temps requerit per els productes de matrius.

En la taula 7.1 es poden veure els resultats dels tests. Per cada conjunt de 10 proves amb els mateixos valors d'n i k s'ha calculat la mitjana del temps. Així, la taula mostra per cada configuració (n, k) la mitjana del temps d'execució $\bar{t}_{n,k}$.

Observant la taula, podem veure com en tots els casos s'ha obtingut una millora considerable. En els tests més intensius s'ha aconseguit gairebé accelerar per 8 el temps fent servir els 8 nodes, arribant molt a prop de la millora màxima que estableix la llei d'Amdahl.

La figura 7.1 mostra els resultats de la taula 7.1 en forma gràfica. S'hi pot comprovar com, a mesura que augmentem la dimensió de les matrius, el temps d'execució creix de forma cúbica. Recordi's que el cost en temps de l'algoritme del producte de matrius és $O(n^3)$.

Un paràmetre interessant d'avaluar és l'*speedup*, que mesura el factor de millora en temps de càlcul per a un k determinat respecte de l'execució seqüencial, és a dir $\frac{\bar{t}_{n,k}}{\bar{t}_{n,1}}$. La taula 7.2 mostra l'*speedup* que obtenim quan $k = 8$ per a diferents valors d'n.

L'únic valor anòmal que es pot veure correspon al cas $n = 24$. Com podem veure a la figura 7.2 el temps de càlcul es redueix aproximadament a la meitat per cada nou node treballador afegit,

Speedup						
$n = 24$	$n = 48$	$n = 96$	$n = 192$	$n = 240$	$n = 384$	$n = 432$
3,609	6,658	7,057	7,773	7,828	7,784	7,757

Taula 7.2: *Speedup* per $k = 8$.

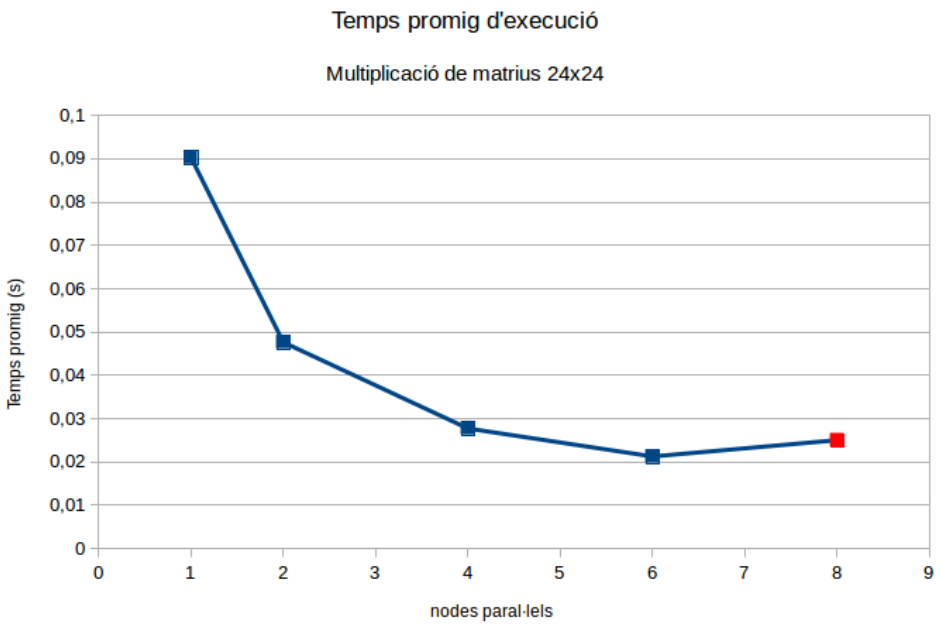


Figura 7.2: Gràfica de $\bar{t}_{n,k}$ per $n = 24$

fins que s'insereix el vuitè, quan el temps augmenta. Els resultats indiquen que es millor utilitzar 6 nodes que 8.

Aquest és un fenomen senzill d'explicar: les funcionalitats MPI requereixen d'un temps per que es produeixi la comunicació. Com més nodes s'involucren, més temps es necessita per aquesta tasca, ja que s'han d'enviar més missatges. Si en un programa el codi paral·lel que ha d'executar cada node requereix poc temps, pot ésser que el temps de la comunicació acabi sent el que domina. Llavors, augmentar el nombre de nodes no millora el resultat.

El master també executa un tros petit de codi per acabar d'obtenir la resposta, això també influeix.

Si ens ho mirem més en detall, podem definir el temps d'execució del programa com la suma del temps dedicat a la tasca paral·lela i el temps dedicat a la resta de funcions (comunicació i master).

$$t_e = t_p + t_a$$

Mitjançant la paral·lelització només podem millorar t_p. Si anem afegint nodes, reduirem de forma progressiva t_p però t_a sempre seguirà present, el temps d'execució del nostre codi mai pot

Grau de paral·lelització						
$n = 24$	$n = 48$	$n = 96$	$n = 192$	$n = 240$	$n = 384$	$n = 432$
0.826	0.971	0.980	0.996	0.997	0.996	0.996

Taula 7.3: Grau de paral·lelització del programa per $k = 8$.

Dispositiu	CPU	Freq. (GHz)	Cost (€)
Portàtil	Intel Core i5-8250U	$1,6$–$3,4$	750,0
Raspberry Pi 3 B+	Broadcom BCM2837B0, Cortex-A53	$1,4$	34,0

Taula 7.4: Característiques del portàtil i de la Raspberry Pi.

disminuïr de t_a. A més, com s'ha dit abans, t_a pot arribar a crèixer.

Això ve a ser també el que ens diu la llei d'Amdahl, si el temps de la tasca paral·lela es redueix tant que passa a ser comparable amb el temps dedicat a altres tasques, com per exemple la comunicació, significa que només part del programa es troba paral·lelitzat i per tant ja no podem aconseguir l'*speedup* màxim. El percentatge de paral·lelització del que parla la llei d'Amdahl no és només un factor de codi, també ho és de temps.

Amb la mateixa llei d'Amhahl podem trobar quin ha estat el percentatge de paral·lelització p per els tests.

$$s = \frac{1}{1 - p + \frac{p}{k}}$$

Si aïllem p:

$$p = \frac{\frac{k}{s} - k}{1 - k}$$

En la taula 7.3 s'observa aquest valor per els diferents tests $k = 8$. Es veu com per el test problemàtic només un 82.6% del temps de programa s'ha dedicat a la tasca paral·lela.

El gràfic de la figura 7.1 també inclou la corba de temps corresponent a una execució seqüencial en un computador portàtil. La taula 7.4 compara les característiques dels nodes del *clúster.* i del portàtil.

El temps d'execució en el portàtil permet comparar el rendiment del *clúster* amb el d'una computador portàtil convencional actual. Noti's com, tot i haver-se reduït molt el temps d'execució, no arriba a la capacitat computacional del portàtil. S'hi apropa molt, però. Cal dir també que el preu total del clúster és inferior al del portàtil, amb un parell de raspberries més potser hauríem aconseguit un rendiment semblant a un mateix preu.

La taula 7.5 mostra el detall dels temps d'execució obtinguts pel portàtil.

$\bar{t}_{n,k}$ per $k = 1$ (s)						
$n = 24$	$n = 48$	$n = 96$	$n = 192$	$n = 240$	$n = 384$	$n = 432$
0,008	0,049	0,368	2,916	5,797	24,751	36,350

Taula 7.5: Temps del producte de matrius seqüencial en un portàtil.

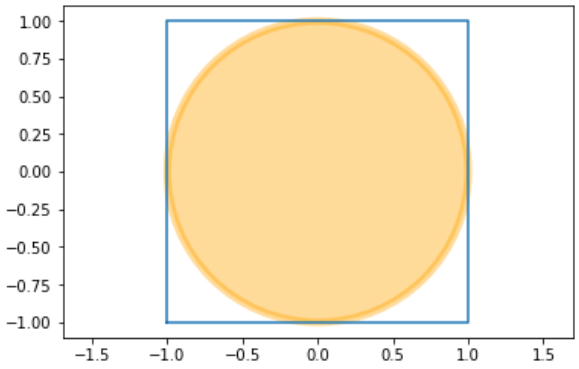


Figura 7.3: Base del test de Monte Carlo per calcular pi.

7.2 Càlcul de Pi a Jupyter. Mètode de Monte Carlo

7.2.1 Descripció del test

És coneix com a mètode de Monte Carlo a tot aquell algoritme que, mitjançant una generació repetitiva de dades aleatories, obté algun resultat numèric. En aquest test s'utilitza un algoritme d'aquest tipus per acabar descobrint el valor de Pi.

Tal i com s'explica a l'extracte *The estimation of Pi using Monte Carlo technique*, [15], la teoria de com s'aconsegueix això comença amb visualitar una circumferència d'una unitat de radi, centrada a l'origen de coordenades i encapsulada dins un quadrat, tal i com es mostra a la figura 7.3.

Sabem que l'àrea del quadrat és $A_q = l^2$ i l'àrea de la circumferència és $A_c = \pi r^2$. Com que $r = \frac{l}{2}$,

$$A_c = \pi r^2 = \pi(\frac{l}{2})^2 = \frac{\pi l^2}{4}$$

i per tant

$$\frac{A_c}{A_q} = \frac{\frac{\pi l^2}{4}}{l^2} = \frac{\pi l^2}{4l^2} = \frac{\pi}{4}$$

Així doncs,

$$\pi = 4\frac{A_c}{A_q}$$

El mètode de Monte Carlo consisteix en anar «llençant» punts aleatòriament distribuits dins el quadrat que s'ha definit i llavors comprovar quants d'aquests punts es troben dins la circumferència. Si la distribució dels punts aleatoris és uniforme (vegeu la figura 7.4), la relació entre

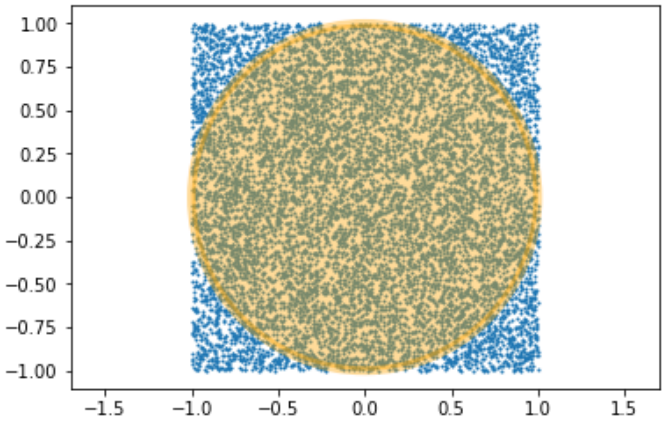


Figura 7.4: Exemple de test de Monte carlo per trobar pi utilitzant 10.000 punts.

punts dins el cercle i punts totals és aproximadament igual a la relació entre l'àrea del cercle i la del quadrat, i per tant s'aproxima a π.

$$\pi = 4\frac{A_c}{A_q} \approx 4\frac{Punts\ dins}{Punts\ totals}$$

El punt clau d'aquest experiment és que com més punts es generin més precissió s'obtindrà en la resposta, ja que es té en compte una part cada cop major de l'àrea. Això, però, també suposa un increment del temps requerit per el test, que reduirem amb la paral·lelització.

7.2.2 Resultats

Una implementació del test la podem dividir en les següents parts:

1. Generar n parells de punts (x, y).
2. Verificar quants d'aquests punts es troben dins el cercle.
3. Calcular el valor estimat de π.

Podem paral·lelitzar molt fàcilment les dues primeres parts de l'algoritme. En lloc de tenir una sola màquina encarregant-se de n punts podem tenir k màquines cadascuna d'elles generant i comprovant $\frac{n}{k}$ punts.

Aquest test s'ha realitzat via `jupyter` i `mpi4py`, per demostrar què podem fer amb aquestes eines. Podem trobar el *notebook* amb tot el codi a l'annex 2.

En programar aquest test, s'ha hagut d'anar amb compte són els *ranks* dels processos i el comunicador de MPI. En utilitzar `ipyparallel`, i per posar en marxa els motors, s'ha engegat ja un procés MPI a cadascún dels nodes. De forma que la mida del comunicador global és sempre 9, el master i el 8 nodes esclau. Si volem executar el codi en un número menor de nodes hem de treballar amb alguna de les funcions per dividir comunicadors d'MPI, de forma que, dins el programa, es creei un comunicador nou amb tants nodes com es desitji. Llavors caldrà adaptar el codi perquè treballi amb aquest comunicador.

	$\bar{t}_{n,k}$ (s)							
k	$n=5\times10^5$	$n=1\times10^6$	$n=1,5\times10^6$	$n=2\times10^6$	$n=2,5\times10^6$	$n=3\times10^6$	$n=3,5\times10^6$	$n=4\times10^6$
1	6,257	12,20	18,207	23,639	30,162	36,020	41,374	47,350
2	3,256	6,243	9,251	12,152	15,215	18,230	21,146	24,054
3	2,257	4,251	6,25	8,242	10,287	12,265	14,208	16,142
4	1,776	3,308	4,754	6,257	7,769	9,255	10,779	12,155
5	1,493	2,691	3,911	5,054	6,254	7,484	8,678	9,823
6	1,341	2,323	3,330	4,339	5,292	6,323	7,370	8,422
7	1,190	2,030	2,925	3,800	4,608	5,476	6,341	7,136
8	1,095	1,853	2,576	3,351	4,100	4,838	5,546	6,273

Taula 7.6: Temps d'execució del test de Monte Carlo en el clúster

	Speedup							
k	$n=5\times10^5$	$n=1\times10^6$	$n=1,5\times10^6$	$n=2\times10^6$	$n=2,5\times10^6$	$n=3\times10^6$	$n=3,5\times10^6$	$n=4\times10^6$
1	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
2	1,921	1,956	1,968	1,945	1,982	1,976	1,957	1,968
3	2,772	2,871	2,913	2,868	2,932	2,937	2,912	2,933
4	3,522	3,690	3,830	3,778	3,882	3,892	3,838	3,895
5	4,191	4,536	4,655	4,676	4,823	4,813	4,767	4,820
6	4,665	5,255	5,468	5,448	5,699	5,696	5,613	5,622
7	5,255	6,013	6,223	6,220	6,545	6,577	6,525	6,635
8	5,714	6,586	7,066	7,053	7,355	7,445	7,460	7,548

Taula 7.7: *Speedups* aconseguits en el *clúster* per Monte carlo

Per l'experiment s'ha seguit un mètode similar a l'anterior. S'han realitzat tests amb una generació de n punts aleatoris on $n = 500000, 1000000, 1500000, 2000000, 2500000, 3000000, 3500000, 4000000$. Cadascuna d'aquestes iteracions s'ha realitzat amb un nombre diferent de nodes en el *clúster* $k = 1, 2, 3, 4, 5, 6, 7, 8$. De nou, cada test s'ha repetit 10 vegades per obtenir resultats més precisos.

Els resultats els podem trobar en les taules 7.6, 7.7 i 7.8, on es mostren els temps mitjans, els *speedups* i els errors en els valors de pi calculats, respectivament.

Ens trobem amb la mateixa situació que en la prova anterior. Els tests més senzills presenten un *speedup* menor que els complexos. La part paral·lela és suficientment ràpida que la resta del codi

	$\bar{e}_{n,k}$							
k	$n=5\times10^5$	$n=1\times10^6$	$n=1,5\times10^6$	$n=2\times10^6$	$n=2,5\times10^6$	$n=3\times10^6$	$n=3,5\times10^6$	$n=4\times10^6$
1	$0,6\times10^{-4}$	$-5,8\times10^{-5}$	$9,8\times10^{-5}$	$5,9\times10^{-5}$	$4,6\times10^{-5}$	$-7,5\times10^{-6}$	$-1,4\times10^{-4}$	$-1,6\times10^{-4}$
2	$-4,3\times10^{-4}$	$-2,4\times10^{-4}$	$2,8\times10^{-4}$	$-4,6\times10^{-5}$	$8,5\times10^{-5}$	$1,4\times10^{-4}$	$4,6\times10^{-5}$	$-1,1\times10^{-4}$
3	$3,1\times10^{-4}$	$4,0\times10^{-5}$	$-9,6\times10^{-5}$	$5,5\times10^{-5}$	$-4,3\times10^{-5}$	$6,1\times10^{-5}$	$1,3\times10^{-5}$	$-3,5\times10^{-5}$
4	$2,4\times10^{-4}$	$-3,8\times10^{-5}$	$4,5\times10^{-5}$	$1,2\times10^{-4}$	$-5,6\times10^{-5}$	$5,0\times10^{-5}$	$5,2\times10^{-5}$	$1,1\times10^{-4}$
5	$0,2\times10^{-4}$	$-1,2\times10^{-4}$	$-8,8\times10^{-5}$	$-7,5\times10^{-5}$	$1,4\times10^{-4}$	$-5,5\times10^{-5}$	$5,0\times10^{-6}$	$1,2\times10^{-4}$
6	$1,1\times10^{-4}$	$-3,3\times10^{-4}$	$2,3\times10^{-5}$	$6,9\times10^{-5}$	$2,2\times10^{-5}$	$5,6\times10^{-5}$	$-4,0\times10^{-5}$	$-2,2\times10^{-5}$
7	$0,9\times10^{-4}$	$-8,4\times10^{-5}$	$-2,2\times10^{-4}$	$-7,4\times10^{-5}$	$5,4\times10^{-5}$	$-8,5\times10^{-5}$	$8,0\times10^{-5}$	$-2,4\times10^{-5}$
8	$-0,5\times10^{-4}$	$4,8\times10^{-5}$	$-3,3\times10^{-5}$	$-8,2\times10^{-5}$	$-1,8\times10^{-4}$	$-9,5\times10^{-5}$	$2,6\times10^{-4}$	$1,3\times10^{-4}$

Taula 7.8: Errors mitjans en l'aproximació de π per Monte Carlo

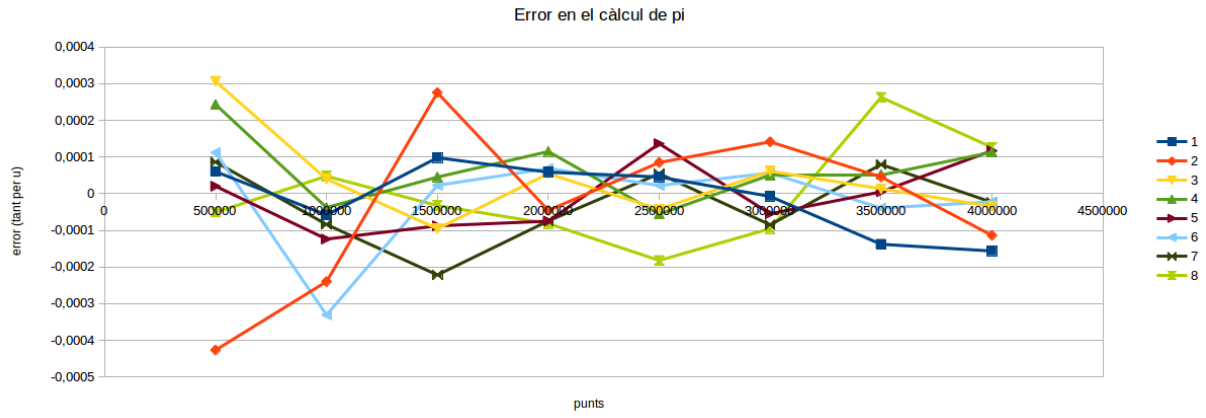


Figura 7.5: Evolució de l'error en incrementar el nombre de punts en el test de Monte Carlo.

acaba representant un percentatge apreciable dins el programa. Podem treure una conclusió de tot això, com més temps requereixi un programa per executar-se, més bons seran els resultats de la seva paral·lelització.

A les dades també s'observa que, tot i el que diu la teoria, no s'ha obintgut cap millora en l'error de π en augmentar el nombre de punts. De fet, si observem el grafic de la figura 7.5 veiem que l'error té un comportament bastant aleatori. A [15] s'apunta com a causa el fet que el mètode de Monte Carlo presenta sempre un error estàndard de $\frac{1}{\sqrt{n}}$. Per valors petits d'n l'error decreix ràpidament. En canvi, a mida que n es fa més gran l'error disminueix més lentament i es necessita afegir una gran quantitat de punts per aconseguir un canvi apreciable en l'error.

7.3 Càlcul de Pi mitjançant Octave mpi

En aquest últim test es calcula el valor de π mitjançant la discretització d'una integral. Un procés fàcilment paral·lelitzable. S'utilitza `octave` com a eina principal.

Descripció del test

Tal i com diu Sung Bae, [3], π es pot calcular amb la següent integral:

$$\pi = \int_0^1 \frac{4}{(1+x^2)} dx$$

Aquest valor el podem aproximar com una suma finitia d'àrees rectangulars de la forma següent:

$$\pi \approx \sum_{n=0}^{N} \frac{4}{(1+x^2)} dx$$

Si fem les àrees més primes; augmenten el nombre de passos, fent el càlcul més costos, però aconseguint un resultat més precís. En aquest problema l'execució paral·lela ens pot ser molt

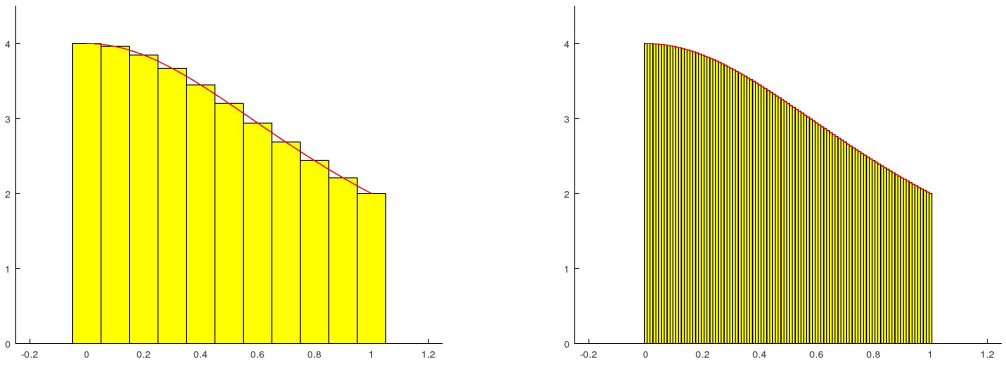


Figura 7.6: Aproximació de la integral amb més o menys rectangles..

útil ja que després de triar el nombre n de passos a realitzar, els podem distribuir en diferents nodes del nostre *clúster*. Cada procés paral·lel calcula un tros de l'àrea i un cop tots han acabat es sumen els seus resultats per obtenir π.

Resultats

Per aquest test d'ha adaptat el programa `Pi.m` que vé inclòs com a exemple dins el paquet MPI d'`Octave`, i que fa exactament el que s'ha descrit en l'apartat anterior.

S'han executat proves en $k = 1, 2, 3, 4, 5, 6, 7, 8$ nodes i separant la integral en n rectangles, $n = 1000000, 10000000, 20000000$. Podem observar els resultats en les taules 7.9, 7.10 i 7.11. En elles es mostren els temps mitjans, els *speedups* i el grau de paral·lelització que s'obté en aplicar la llei d'Amdahl a les dades.

Els resultats no són massa positius, de fet, s'han aconseguit millores de temps molt pobres. Sin ens fixem en els graus de paral·lelització podem veure que, en alguns casos, aquests donen valors molt estranys i que varien en canviar els número de nodes.

Amdahl diu amb la seva llei que el fet d'afegir nodes redueix progressivament el temps d'execució, fins que s'arriba a un moment en que ja no es pot obtenir cap més milora, sempre i quan el programa no estigui 100% paral·lelitzat. El que no diu mai és que afegir nodes pugui incrementar el temps, com passa amb les nostres dades.

Cal dir que Amdahl només atribuia el percentatge de paral·lelització al codi i, com ja s'ha vist en el test anterior, pot ser que a l'hora d'executar un programa diferents factors canviin el balanç temporal entre la part paral·lela i la resta del codi, modificant efectivament aquest percentatge de paral·lització teòric, fent que el temps de la part seqüencial creixi. En el codi utilitzat es pot veure com es llegeixen les respostes dels nodes de forma seqüencial, aquest fet pot ser un justificant del que s'ha explicat.

MPI ofereix mètodes especials per recopilar les dades dels nodes, utilitzar-ne algún podria suposar una gran millora en el codi.

	$\bar{t}_{n,k}$ (s)		
k	$n = 10000000$	$n = 10000000$	$n = 20000000$
1	0,248	0,702	1,211
2	0,245	0,597	0,864
3	0,233	0,464	0,708
4	0,261	0,42	0,624
5	0,263	0,397	0,572
6	0,472	0,463	0,502
7	0,454	0,591	0,753
8	0,663	0,773	0,888

Taula 7.9: Temps d'execució mitjà en el càlcul de π amb `octave`

	Speedup		
k	$n = 10000000$	$n = 10000000$	$n = 20000000$
1	1	1	1
2	1,011	1,177	1,401
3	1,061	1,514	1,71
4	0,948	1,673	1,941
5	0,942	1,768	2,117
6	0,525	1,518	2,412
7	0,546	1,189	1,607
8	0,374	0,909	1,364

Taula 7.10: *Speedup* en el càlcul de π amb `octave`.

	Grau de paral·lelització		
k	$n = 10000000$	$n = 10000000$	$n = 20000000$
1	0	0	0
2	0,022	0,301	0,572
3	0,087	0,509	0,623
4	-0,073	0,537	0,647
5	-0,077	0,543	0,66
6	-1,084	0,409	0,703
7	-0,971	0,186	0,441
8	-1,916	-0,114	0,305

Taula 7.11: Grau de paral·lelització segons Amdahl en el càlcul de π amb `octave`.

8 Conclusions

Aquest ha estat un projecte que en alguns punts s'ha complicat més del esperat, principalment en la configuració d'alguns dels softwares. Considero però, que els objectius proposats s'han complert en un bon nivell. El punt negatiu és que alguns d'ells d'aquests objectius s'han hagut de canviar lleugerament.

En un principi és tenia la intenció d'executar en el *clúster* algun programa més complex que ens donés resultats més visuals, com per exemple algun software de simulació científica o de renderitzat d'imatges. Per restriccions de temps s'ha agafat una alternativa més segura i s'ha orientat el projecte a utilitats de programació paral·lela. El fet d'utilitzar Raspberry Pis també ha influït en aquest tema, molts dels programes existents no són «amigables» amb l'arquitectura ARM, cosa que ens dificulta o impossibilita la instal·lació.

`Octave` també ha estat un tema crític. Aquest és un software que durant la carrera hem utilitzat bastant, així que hem feia gràcia incloure'l en el treball. Desafortunadament la poca informació sobre el paquet MPI i les incompatibilitats entre versions han posat les coses difícils. Al final hi ha hagut èxit en paral·lelitzar `Octave`, però s'hi ha hagut de dedicar bastant temps extra.

Tot i aixi, hem pogut obtenir un *clúster* funcional i hem realitzat en ell proves satisfactòries. Cal destacar el rendiment aconseguit ja que ens les proves gairebé hem obtingut la màxima millora de velocitat teòrica.

El projecte també ha servit per conèixer més del món de la programació paral·lela. Un tema que en el transcurs de la carrega ja s'havia vist un parell de cops, principalment amb erlang. MPI ha estat un tecnologia que ha complementat molt bé aquests coneixements, a més, tot el software carregat a sobre d'MPI ha donat una visió de com d'útil pot ser aquesta eina.

Acrònims i Abreviatures

A

amd Advanced Microdevices Inc.. 8

api Application programming interface. 8

arpanet The Advanced Research Projects Agency Network. 4

B

bsc Barcelona Supercomputing Center. 11

C

cdc Control Data Corporation. 9

cesdis Center of Excellence in Space Data and Information Sciences. 5

D

dec Digital Equipment Corporation. 4

E

ess Earth and Space Sciences. 5

G

gpu Graphics processing unit. 11, 12

H

hpc High Performance Computing. 9

I

ibm International Business Machines. 3, 7, 8

ipython Interactive Python. 21, 22, 23

ipyparallel IPython Parallel. 22

M

mpi Message Passing Interface. vii, 5, 6, 7, 8, 9, 12, 15, 19, 20, 22, 27, 28, 29, 31, 32, 33, 34, 37, 38, 45, 47, 50, 53, 55, 61

N

nasa The National Aeronautics and Space Administration. 5

nersc National Energy Research Scientific Computer Center. vii, 9

nfs Network Filesystem. 28, 29, 30

O

openmp Open Multiprocessing. 8, 9

openmpi Open Message Passing Interface. 8, 31, 32, 33, 37, 38, 39

ornl Oak Ridge National Laboratory. 11

P

pvm Parallel Virtual Machine. 5, 7

R

rpi Raspberry Pi. 17, 18, 19, 32, 35

S

ssh Secure Shell. 29, 30, 31, 34

T

tcp/ip Transmission Control Protocol/Internet protocol. 4

Bibliografia

[1] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". A: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS'67 (Spring). Atlantic City, New Jersey: ACM, 1967, pàg. 483 - 485. DOI: 10.1145/1465482.1465560. URL: http://doi.acm.org/10.1145/1465482.1465560.

[2] R&D 100 Conference & Awards. *rd100 awards*. 2018. URL: https://www.rd100conference.com/.

[3] Sung Bae. *A Hands-on Introduction to MPI Python Programming*. Inf. tèc. New Zealand eScience Infrastructure. University of Auckland, 2018. URL: https://www.nesi.org.nz/sites/default/files/mpi-in-python.pdf (cons. 21-4-2018).

[4] Gordon Bell. *Supercomputers: The Amazing Race. A History of Supercomputing, 1960-2020*. TechReport MSR-TR-2015-2. Ver. 1. 555 California, 94104 San Francisco, CA: Microsoft Corporation, nov. de 2014. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2015/01/MSR-TR-2015-2_Supercomputers-The_Amazing_Race_Bell.pdf (cons. 20-6-2018).

[5] MPICH Contributors. *MPICH Derivative Implementations*. 2018. URL: https://www.mpich.org/about/collaborators/ (cons. 21-4-2018).

[6] OpenMPI Contributors. *The Open MPI project contributors*. 2018. URL: https://www.open-mpi.org/about/members.

[7] NVIDIA Corporation, ed. *Introducing the world's most powerful supercomputer. A new age of scientific discovery*. 2018. URL: https://images.nvidia.com/content/pdf/worlds-fastest-supercomputer-summit-infographic.pdf (cons. 4-7-2018).

[8] Lisandro Dalcin. *MPI for Python*. Ver. 3.0.0. 8 de nov. de 2017. URL: http://mpi4py.scipy.org/docs/mpi4py.pdf (cons. 18-6-2018).

[9] James R. Fischer. "The Roots of Beowulf". A: *Proceedings of the 20 Years of Beowulf Workshop on Honor of Thomas Sterling's 65th Birthday*. Beowulf '14. Annapolis, MD, USA: ACM, 2015, pàg. 1 - 6. ISBN: 978-1-4503-3031-2. DOI: 10.1145/2737909.2770195. URL: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20150001285.pdf (cons. 2-7-2018).

[10] Raspberry Pi Foundation. *Raspberry Pi - Downloads*. 2018. URL: https://www.raspberrypi.org/downloads.

[11] A. Geist et al. *PVM: Parallel Virtual Machine: a Users Guide and Tutorial for Networked Parallel Computing*. Scientific and engineering computation. MIT Press, 1994. ISBN: 9780262571081. URL: https://books.google.es/books?id=afjci8AOy6EC.

[12] Kal Hwang, Jack Dongarra i Geoffrey Fox. "Virtualization: Physical vs. Virtual Clusters. Deploying multiple virtual machines into clusters requires some special management and configuration techniques". A: *Microsoft Technet Magazin* (abr. de 2012). Adaptat de «Distributed and Cloud Computing: From Parallel Processing to the Internet of Things», Syngress 2011. URL: `https://technet.microsoft.com/en-us/library/hh965746.aspx` (cons. 2-7-2018).

[13] Shirley Moore. *Introduction to OpenMP*. Course Notes. University of Tennessee, 9 de mar. de 2011. URL: `http://www.netlib.org/utk/people/JackDongarra/WEB-PAGES/SPRING-2011/Lect09-OpenMP.pdf` (cons. 28-5-2018).

[14] Gregory F. Pfister. *In Search of Clusters*. 2a ed. Prentice-Hall, Inc., 1998. ISBN: 0-13-899709-8.

[15] Svetlana Strbac-Savic, Ana Miletic i Hana Stefanović. "The estimation of Pi using Monte Carlo technique with interactive animations". A: *Proc. of the 8th International Scientific Conference "Science and Higher Education in Function of Sustainable Development - SED 2015*. In CD support. Uzice, Serbia, oct. de 2015. URL: `https://www.researchgate.net/publication/282909462` (cons. 26-6-2018).

[16] The IPython Development Team. *IPython Parallel Documentation*. Ver. 6.3.0.dev. 12 de gen. de 2018. URL: `https://media.readthedocs.org/pdf/ipyparallel/latest/ipyparallel.pdf` (cons. 17-6-2018).

[17] Jon W. Eaton et al. *GNU Octave. A high-level interactive language for numerical computations*. 4a ed. 1 d'abr. de 2018. 1041 pàg. URL: `https://octave.org/octave.pdf` (cons. 28-5-2018).

Annexes

1 Producte de matrius amb mpi4py

Producte de matrius paral·lel per a una arquitectura MPI escrit amb `Python` emprant el mòdul `mpi4py`.

```
1  #!/usr/bin/env python3
2
3  """
4  @Author: Jordi Corbilla
5  @Description: Parallel MPI Matrix Multiplication (NxN)
6
7  This program is free software: you can redistribute it and/or modify
8  it under the terms of the GNU General Public License as published by
9  the Free Software Foundation, either version 3 of the License, or
10 (at your option) any later version.
11
12 This program is distributed in the hope that it will be useful,
13 but WITHOUT ANY WARRANTY; without even the implied warranty of
14 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
15 GNU General Public License for more details.
16
17 You should have received a copy of the GNU General Public License
18 along with this program.  If not, see <http://www.gnu.org/licenses/>.
19 """
20
21 from mpi4py import MPI
22 import sys
23 import numpy as np
24
25 numberRows = int( sys.argv[1])
26 numberColumns = int( sys.argv[2])
27 TaskMaster = 0
28
29 assert numberRows == numberColumns
30
31 #print ("Initialising variables.\n")
32 a = np.zeros(shape=(numberRows, numberColumns))
33 b = np.zeros(shape=(numberRows, numberColumns))
34 c = np.zeros(shape=(numberRows, numberColumns))
35
36 def populateMatrix( p ):
37     for i in range(0, numberRows):
38         for j in range(0, numberColumns):
39             p[i][j] = i+j
40
41 comm = MPI.COMM_WORLD
42 worldSize = comm.Get_size()
43 rank = comm.Get_rank()
```

```
44 processorName = MPI.Get_processor_name()
45
46 #print ("Process %d started.\n" % (rank))
47 #print ("Running from processor %s, rank %d out of %d processors.\n" % (processorName, rank, worldSize
       ))
48
49 #Calculate the slice per worker
50 if (worldSize == 1):
51     slice = numberRows
52 else:
53     slice = numberRows / (worldSize-1) #make sure it is divisible
54 assert slice%1==0
55 assert slice >= 1
56 slice = int(slice)
57
58 populateMatrix(b)
59
60 comm.Barrier()
61
62 if rank == TaskMaster:
63     #print ("Initialising Matrix A and B (%d,%d).\n" % (numberRows, numberColumns))
64     #print ("Start")
65     populateMatrix(a)
66     t_start = MPI.Wtime()
67     for i in range(1, worldSize):
68         offset = (i-1)*slice #0, 10, 20
69         row = a[offset,:]
70         comm.send(offset, dest=i, tag=i)
71         comm.send(row, dest=i, tag=i)
72         for j in range(0, slice):
73             comm.send(a[j+offset,:], dest=i, tag=j+offset)
74     #print ("All sent to workers.\n")
75
76 comm.Barrier()
77
78 if rank != TaskMaster:
79
80     #print ("Data Received from process %d.\n" % (rank))
81     offset = comm.recv(source=0, tag=rank)
82     recv_data = comm.recv(source=0, tag=rank)
83     for j in range(1, slice):
84         c = comm.recv(source=0, tag=j+offset)
85         recv_data = np.vstack((recv_data, c))
86
87     #print ("Start Calculation from process %d.\n" % (rank))
88
89     #Loop through rows
90     t_start = MPI.Wtime()
91     for i in range(0, slice):
92         res = np.zeros(shape=(numberColumns))
93         if (slice == 1):
94             r = recv_data
95         else:
96             r = recv_data[i,:]
97         ai = 0
98         for j in range(0, numberColumns):
99             q = b[:,j] #get the column we want
100            for x in range(0, numberColumns):
101                res[j] = res[j] + (r[x]*q[x])
```

```
102             ai = ai + 1
103         if (i > 0):
104             send = np.vstack((send, res))
105         else:
106             send = res
107     t_diff = MPI.Wtime() - t_start
108 
109     #print("Process %d finished in %5.4fs.\n" %(rank, t_diff))
110     #Send large data
111     #print ("Sending results to Master %d bytes.\n" % (send.nbytes))
112     comm.Send([send, MPI.FLOAT], dest=0, tag=rank) #1, 12, 23
113 
114 #comm.Barrier()
115 
116 if rank == TaskMaster:
117     #print ("Checking response from Workers.\n")
118     res1 = np.zeros(shape=(slice, numberColumns))
119     comm.Recv([res1, MPI.FLOAT], source=1, tag=1)
120     #print ("Received response from 1.\n")
121     kl = np.vstack((res1))
122     for i in range(2, worldSize):
123         resx= np.zeros(shape=(slice, numberColumns))
124         comm.Recv([resx, MPI.FLOAT], source=i, tag=i)
125         #print ("Received response from %d.\n" % (i))
126         kl = np.vstack((kl, resx))
127     #print ("End")
128     #print ("Result AxB.\n")
129     #print (kl)
130     t_diff = MPI.Wtime() - t_start
131     #print("Process %d received the result in %5.4fs.\n" %(rank, t_diff))
132     print("%d,%d,%5.4fs" % (numberRows,worldSize,t_diff))
133 comm.Barrier()
```

2 Notebook jupyter per calcular pi amb Monte Carlo

Definició del programa

Escrivim un programa per executar l'algoritme amb mpi4py. Permet triar els punts a generar i el nombre de nodes paral·lels que volem

In [6]:
```
%%writefile montecarlopi2.py
from mpi4py import MPI
import random as rdm
import math

def run_task(punts, nodes=None):
    world_comm = MPI.COMM_WORLD
    world_rank = world_comm.rank

    if nodes <=0:
        return -1
    if nodes:
        #Creem un nou comunicador amb tants nodes com es volen
        triat = MPI.UNDEFINED
        if world_rank <= nodes:
            triat = True
        comm = world_comm.Split(triat,world_rank)
        if triat == MPI.UNDEFINED:
            return ([],[])
    else:
        #Treballem amb el comunicador global (tots els nodes)
        comm = world_comm

    rank = comm.rank
    participants = comm.size
    particio = punts/(participants-1)
    residu = punts%(participants-1)

    dins=0 #Punts dins el cercle
    xs=[]
    ys=[]

    if rank != 0:
        #L'últim node realitza els punts residuals
        if rank == participants-1:
            particio+=residu
        for i in range(int(particio)):
            #Punts aleatoris en el primer quadrant
            x=rdm.uniform(-1,1)#random()**2
            y=rdm.uniform(-1,1)#random()**2
            #xs+=[x]
            #ys+=[y]
            #Comprovem si el punt és dins el cercle
            if math.sqrt(x**2+y**2)<1.0:
                dins+=1

    #Recollim la suma al master
    suma = comm.reduce(dins,op=MPI.SUM,root=0)
    if rank == 0:
        pi=4.0*suma/punts
        return pi
    return #(xs,ys)
```

```
Overwriting montecarlopi2.py
```

Connexió amb el cluster

In [1]:
```
import ipyparallel as ipp
import time
```

In [2]:
```
c = ipp.Client(profile='cluster_rpi_mpi')
```

In [3]:
```
c.ids
```

Out[3]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

In [4]:
```
vista=c[:]
vista.targets
```

Out[4]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

In [5]:
```
#Activem la vista per poder executar Magics de Jupyter (%%) amb ella
vista.activate
```

Out[5]:
```
<bound method DirectView.activate of <DirectView [0, 1, 2, 3,...]>>
```

Execució del test

In [7]:
```
#Xecutem el fitxer en els motors
vista.run("montecarlopi2.py")
```

Out[7]:
```
<AsyncResult: execute>
```

In [8]:
```
import numpy
```

In [9]:
```
%%capture dades --no-stderr
print("nodes;punts;resultat;error;temps")
for punts in [500000,1000000,1500000,2000000,2500000,3000000,3500000,4000000]: # Número de punts a generar
    for nodes in [1,2,3,4,5,6,7,8]:               # Nodes paral·lels
        for j in range(10):                       # 10 repeticions per cada test
            start = time.time()
            vista["punts"] = punts
            vista["nodes"] = nodes
            %px result = run_task(punts,nodes)
            end = time.time()
            temps = end-start
            resultat = vista["result"][0]
            error = (1-(resultat/numpy.pi))
            print("%d;%d;%f;%f;%f" % (nodes,punts,resultat,error,temps))
```

In [10]:
```
#Escrivim els resultats en un fitxer de text
with open('resultats_monte_carlo2.txt', 'w') as f:
    f.write(dades.stdout)
```

3 Programa d'Octave per calcular pi amb mpi

```
function result = Pi2String (N, mod)
  ## ArgChk
  if (nargin < 1)
    N = 1E7;
  end
  if (nargin < 2)
    mod = 's';
  end
  if (nargin > 2)
    print_usage (); # let all ranks complain
  end
  flag = 0; # code much simpler
  flag = flag || ~isscalar (N) || ~isnumeric (N);
  flag = flag || fix(N) ~= N || N < 1;
  mod = lower (mod); mods = 'sr';
  flag = flag || isempty (findstr (mod, mods));
  if (flag)
    print_usage (); # let them all error out
  end

  ## Results struct
  results.pi = 0;
  results.err = 0;
  results.time = 0;
  result="";

  ## PARALLEL: initialization, include MPI_Init time in measurement
  T=clock;

  MPI_ANY_SOURCE = -1;
  MPI_Init ();
  mpi_comm_world = MPI_COMM_WORLD;
  rnk   = MPI_Comm_rank (mpi_comm_world); # let it abort if it fails
  siz   = MPI_Comm_size (mpi_comm_world);
```

```
35
36  SLV = logical(rnk);                 # handy shortcuts, master is rank 0
37  MST = ~ SLV;                         # slaves are all other
38
39  ## PARALLEL: computation (depends on rank/size)
40  width=1/N; lsum=0;                 # for i=rnk:siz:N-1
41  i=rnk:siz:N-1;                   #   x=(i+0.5)*width;
42  x=(i+0.5)*width;                  #   lsum=lsum+4/(1+x^2);
43  lsum=sum(4./(1+x.^2));           # end
44
45  ## PARALLEL: reduction and finish
46  switch mod
47    case 's',
48      TAG=7;                           # Any tag would do
49      if SLV                           # All slaves send result back
50        MPI_Send (lsum, 0, TAG, mpi_comm_world);
51      else                             # Here at master
52        Sum =lsum;                       # save local result
53        for slv=1:siz-1                  # collect in any order
54          lsum = MPI_Recv (MPI_ANY_SOURCE, TAG, mpi_comm_world);
55          Sum += lsum;                     # and accumulate
56        endfor                           # order: slv or MPI_ANY_SOURCE
57      endif
58    case 'r',
59      disp ("not yet implemented");
60  endswitch
61
62  MPI_Finalize ();
63
64  if MST
65    Sum      = Sum/N ;                    # better at end: don't loose resolution
66    results.time = etime(clock,T);  #       # but only at master after PI computed
67    results.err  = Sum - pi;
68    results.pi   = Sum;
69    # Concatenem els resultats en un string
70    pi=results.pi;
71    err=results.err;
72    time=results.time;
73    result = [num2str(pi) ";" num2str(err) ";" num2str(time)]
74  endif
75
76 endfunction
```